目錄

Kivy中文编程指南

Kivy 是什么？

Kivy 是一个开源的 Python 框架，用于快速开发应用，实现各种当前流行的用户界面，比如多点触摸等等。

Kivy 可以运行于 Windows， Linux， MacOS， Android， iOS 等当前绝大部分主流桌面/移动端操作系统。

Kivy 基于 Python，界面文件和程序文件相互分离的设计思路，设计简洁优雅，语法易学，适合新人入门。

目前 Kivy 的官方文档还不算很完善，中文翻译版本之前也有过，但往往没有进行持续更新，或者内容覆盖不全面，所以我们这次来一个长期持久的跟进。同时期待你的参与！

如果你想参与进来，可以加入我们的 Kivy 交流QQ群号:248136053，也可以关注我的[知乎专栏](https://zhuanlan.zhihu.com/python-kivy)

从去年开始的Kivy 编程指南中文翻译项目，今天基本算是弄完了，把 Kivy Programming Guide 里面的全部内容翻译了一遍。

当然了，质量还是不怎么样好，所以希望大家多批评指正，我会尽快改正。

最开始用 Kivy 的时候，感觉有各种各样的 bug ，觉得安装和配置各种繁琐扯皮麻烦，心情也容易变得特别臭。

然后开始翻译文档的时候，也还是容易经常变得很暴躁，尤其是遇到一些原文的语法错误、逻辑错误、自己鬼扯也扯不通顺的地方等等。

但是后来我逐渐地开始钦佩这些创建了 Kivy 的人们，我意识到他们也跟我一样是一群热爱 Python 的人，想为更多的同样使用 Python 的开发者提供一个完整的工具链，以便于能更简洁轻快地实现跨平台开发。

所以，我觉得 Kivy 是一个因为热爱而驱动去追寻自由的项目，这也让我逐渐对 Kivy 产生了更多的好感。

事到如今我可能并不一定会有机会深入使用 Kivy 去开发，也依然希望 Kivy 能够发展壮大，生根发芽，开花散种，给更多的跟我一样初入门无所适从的新手一个友好又高效的选择。

我对 Kivy 官方文档的翻译，暂告一段落。后面的 API 翻译，有一位知乎上的朋友表示有兴趣进行，一位非常年轻有为的少年。看到现在的青少年人都有如此的学习动力和探索精神，不由得让我感慨时代发展之快，自己奔四的路上，还有幸结识了众多初升的太阳。我相信他们

会在未来闪耀，那个时代一定更美好。

Title: Kivy CN Date: 2017-02-12 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：我为什么对Kivy官方文档进行翻译

从去年开始，我陆续翻译了一些Kivy官方文档中的开发指南的内容，地址在这里。

然后我又觉得有必要找一个更大的平台，以便于能给更多人提供一点便利，所以我又开了一个知乎专栏。

然而新年这一阵，我做了个手术，身体状况不太好，感觉理解能力和表达能力也有所下降（其实本来也不行），所以我就想，不如按照之前ThinkPython的中文翻译那样，直接把翻译稿开源放到Github吧，地址在这里。虽然我并没有想出来这样有什么更好的。

接触Kivy时间不长，一年多前最开始知道有这个项目的，当时的观感很差，因为遇到若干个Bug，反馈了之后也没见他们有什么动静。然后时间长了，发现跨平台的Python除了QT基本就只有这个了，相比之下，这个好歹不那么庞大，还是挺好玩的。但也就这样了，没有进一步关注。

然后是要写GeoPython，一些基础的方法都实现了之后，遇到了一些数学上的问题，然后学了一些数学相关的内容，大概有了解决思路之后，才意识到，TMD没有GUI啊，这样常规的地球科学领域的同行们根本懒得看对不对？日常用户才懒得吭哧吭哧学习如何在Bash或者Powershell之下使用iPython运行某个脚本对不对？所以我需要GUI，然而QT太庞大繁杂了，衡量了一下自己的智力水平，估计至少要花费半年才能大概入门。所以我又捡起来Kivy了。

这个过程中我发现Kivy相关的中文资料还真不多，那我就从最基础的官方文档开始翻译一下吧，好歹自己边学习边相当于做笔记，以教促学，还能给人提供一点有用的参考，哪怕一丁点用处也好。

就像我当年给人辅导研究生的C++和考博英语一样，其实也是给自己的持续学习找一个持久的动力，也是争取有一点能够积攒努力产生一个突破口。就像我的启蒙老师许先生当年给我讲的庞中华老前辈一样，一点一点积累总会有收获。

我这些翻译的水平良莠不齐，其中有些简单的部分，我基本可以直接进行双语转换，这就不费什么力气。而由于我在编程的经验和水平两方面都比较差，有的部分一些术语名词翻译得不伦不类，所以我又只能心虚地标记上英文，避免对读者产生太严重的误导。

但我还是会继续下去的，学习和翻译两个过程还不能停下来。我不能因为自己现在三十多岁了而且水平还很差，就停止学习提高的尝试，因为一旦停下来，就更是一点希望都没有了，那就是直接向命运举白旗投降了。

虽然时代已经不同了，我还是很钦佩王江民老前辈，专注和持久而创造了传奇。我没有那么大的野心，也不奢求什么辉煌成就，只是觉得有生之年，做点不后悔的事情吧。

Title: Kivy Installment Tutorial Date: 2016-12-30 Category: Kivy Tags: Python,Kivy

Kivy 中文安装指南

译者的话： 接触Python有一段时间了，之前翻译过ThinkPython2E，我也仍然还是个很菜很弱很入门的外行人。 我接下来翻译的关于Kivy的各种内容，不出意外的话也必将充满了各种低级错误。 如果这些错误有影响到大家阅读理解，提前表示一下歉意。 特别希望大家能把错误的地方指出来，让我学习的同时也及时改正。 我的编程水平很差，然而我热爱计算机这一工具，所以我喜欢做各种探索； 我的英语水平也差，不过我喜欢从英语世界发现有意思的事物并且分享给中文世界的朋友们。 本次翻译将仅仅翻译官方正式版部分的安装指南，每夜版以及更进阶的安装内容，不做翻译。 为什么呢？ 一来是我自身水平有限，二来你都折腾每夜版了还不好好学英文还要看别人翻译也太不靠谱了对不对？

以下是对 Kivy 官方网站安装文档部分的翻译，下面三个链接是原文地址： Installation on Windows Installation on Linux Installation on Mac OS

Windows系统安装Kivy指南

自从1.9.1版本开始，Kivy 官方提供了二进制 wheels 文件，可以用安装Kivy以及所需的一切依赖包到一个已经安装好的Python环境中，在下文会有讲解。

此外还有每夜版wheels文件，可供用户安装，或者用于将之前安装的Kivy更新到新版本；另外本文的后文中也会讲解如何将Kivy安装到自定义位置，而不是安装到默认的site-packages文件夹。

特别注意

【译者注：这段内容是对官方文档比较忠实的翻译和还原，其中提到的MinGW和Python3.5的兼容问题，翻译者没遇到也没有测试过，因为没有Windows的机器。在Mac和Linux上3.5和3.6都成功安装了Kivy，运行过程也没发现问题。】

目前因为MinGW和Python3.5的兼容问题，在Windows平台上还没有办法通过Python3.5来使用Kivy，至少相当一段时间内是没指望了，更多细节参考这里的这个issue。 想要解决这个问题，需要用MSVC来编译的3.5，不过目前还没能实现，所以如果你搞定了MSVC编译的话一定反馈一下。

必要前提

要使用Kivy，首先就得安装Python。Python有好多版本，你可以同时安装其中的好多个，如果你在其中某一个版本的Python里面要使用Kivy，就要在这个版本里面单独按照一次Kivy，其他版本要使用Kivy需要另外再进行安装，就是说每一次安装Kivy只对一份Python环境有效。

安装过程

安装了Python之后，打开命令行工具cmd，然后按照下面的命令来进行Kivy的安装。

1 首先要保证已经安装了最新的pip和wheel：

```
python -m pip install --upgrade pip wheel setuptools
```

2 然后安装必要的依赖包(其中gstreamer大小接近90MB，如果不需要用，就可以跳过不安装这个包:

```
python -m pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew
python -m pip install kivy.deps.gstreamer --extra-index-url https://kivy.org/downloads
/packages/simple/
```

3 如果上一步都成功了没什么报错，就可以按照Kivy了:

```
python -m pip install kivy
```

4 在环境变量中添加一些路径到PATH来避免遇到各种issues (在你的python.exe所在的路径下运行下面的命令):

```
set PATH=%PATH%;%cd%\share\sdl2\bin;%cd%\share\glew\bin
```

到现在为止就搞定了，你就可以在这份Python环境中通过import kivy命令来导入和使用Kivy了。

特别注意

如果你遇到了**permission denied**或者访问被拒绝之类的错误提示，你可以试试以管理员权限来运行命令行工具cmd。

Linux系统安装Kivy指南

使用包管理器进行安装

本节是给各种发行版用对应的deb或者rpm之类的包进行安装s .deb/.rpm/...

Ubuntu / Kubuntu / Xubuntu / Lubuntu (13.10 Saucy Salamander以及之后更新的版本)

1 首先要根据你的喜好来选择一个PPA源添加到你的系统里：

（译者注：这里稳定版和每夜版二选一就可以，如果要体验最新特性，可以使用每夜版，但是如果用于长期使用追求稳定，推荐用稳定版，二者千万不要同时添加，避免出现混乱和错误。）

```
sudo add-apt-repository ppa:kivy-team/kivy #稳定版
sudo add-apt-repository ppa:kivy-team/kivy-daily  #每夜版
```

2 然后就要用包管理器来更新一下包列表了：

```
sudo apt-get update
```

3 更新列表完毕之后，如果没有错误，就可以安装了：

```
sudo apt-get install python-kivy #Python2 用这个来安装
sudo apt-get install python3-kivy #Python3 要加一个3
sudo apt-get install python-kivy-examples #可选的样例代码
```

Debian (8.0 Jessie或者更新的版本)

特别注意

Debian 7 Wheezy 已经不支持了，你至少要升级到Debian 8 Jessie 才能安装Kivy。

1 通过Synaptic新立得包管理器把下面的PPA源添加到你的sources.list列表中，手动添加也可以：

- Jessie/Testing:

```
#稳定版:

deb http://ppa.launchpad.net/kivy-team/kivy/ubuntu trusty main

#每夜版:

deb http://ppa.launchpad.net/kivy-team/kivy-daily/ubuntu trusty main
```

- Sid/Unstable:

```
#稳定版:

deb http://ppa.launchpad.net/kivy-team/kivy/ubuntu utopic main

#每夜版:
deb http://ppa.launchpad.net/kivy-team/kivy-daily/ubuntu utopic main
```

2 添加了源之后，就是要添加一些GPG key到你的apt keyring里面了，运行下面的命令：

```
#非root用户:
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys A863D2D6

#root用户:
apt-key adv --keyserver keyserver.ubuntu.com --recv-keys A863D2D6
```

3 然后跟Ubuntu里面那个类似，更新列表完毕之后，如果没有错误，就可以安装了：

```
sudo apt-get update #安装之前一定要先这样更新一下列表
sudo apt-get install python-kivy #Python2 用这个来安装
sudo apt-get install python3-kivy #Python3 要加一个3
sudo apt-get install python-kivy-examples #可选的样例代码
```

Mint/Bodhi/Suse/Gentoo 这部分省略

Mint的安装基本与对应的Ubuntu版本一样，Bodhi估计用的人不多我懒得翻译了，OpenSuse/Gentoo 的用户应该比较有经验了还用得着我翻译么？

Fedora

1 在终端里添加repo(注意版本不要弄错):

```
#Fedora 18:
sudo yum-config-manager  --add-repo=http://download.opensuse.org\
/repositories/home:/thopiekar:/kivy/Fedora_18/home:thopiekar:kivy.repo

#Fedora 17:
sudo yum-config-manager --add-repo=http://download.opensuse.org\
/repositories/home:/thopiekar:/kivy/Fedora_17/home:thopiekar:kivy.repo

#Fedora 16:
sudo yum-config-manager --add-repo=http://download.opensuse.org\
/repositories/home:/thopiekar:/kivy/Fedora_16/home:thopiekar:kivy.repo
```

2 跟Ubuntu和Debian里面一样，添加源之后还是要用包管理器更新一下包列表。

3 更新列表之后通过包管理器就可以安装**python-Kivy**和**python-Kivy-examples**了。

在虚拟环境中安装

必备的依赖包

Cython

一定要注意，这里超级重要，不同版本的Kivy只能用特定版本的Cython配合才能使用，二者一定要匹配，关系如下表：

Kivy	Cython
1.8	0.20.2
1.9	0.21.2
1.9.1	0.23

SDL2框架相关的依赖包

以Ubuntu为例

在下面的命令样例中，用**python**和**python-dev**就表明是给**Python2**安装，而**python3**和**python3-dev**是针对**Python3**的。

```
# 先要安装这些必备的组件，注意上面刚提示过，如果要用Python3，在python后面加上一个3就可以了:
sudo apt-get install -y \
python-pip \
build-essential \
git \
python \
python-dev \
ffmpeg \
libsdl2-dev \
libsdl2-image-dev \
libsdl2-mixer-dev \
libsdl2-ttf-dev \
libportmidi-dev \
libswscale-dev \
libavformat-dev \
libavcodec-dev \
zlib1g-dev
```

特别注意

在某些特定的Linux版本上，你可能会收到一个与ffmpeg包有关的错误信息。这种情况下，就可以用libav-tools替换掉上文命令中的ffmpeg，另一种解决方法是使用ppa来安装ffmpeg，使用的命令如下所示：

```
sudo add-apt-repository ppa:mc3man/trusty-media
sudo apt-get update
sudo apt-get install ffmpeg
```

要在虚拟环境中安装Kivy了

首先要确保Pip, Virtualenv 和 Setuptools 这几个包都更新到最新：

```
sudo pip install --upgrade pip virtualenv setuptools
```

然后创建一个名为"kivyinstall"的新的虚拟环境，这时候你有两种选择：

- 第一种是用系统自带的默认Python解释器：

```
virtualenv --no-site-packages kivyinstall
```

- 第二种是设定一个指定位置的Python解释器，这里举例就假设要用的解释器路径在/usr/bin/python2.7

```
virtualenv --no-site-packages -p /usr/bin/python2.7 kivyinstall
```

建立好虚拟环境之后，就是进入里面了：

```
. kivyinstall/bin/activate
```

千万要注意，这里要安装正确的Cython版本：

```
pip install Cython==0.23
```

然后就可以在这个虚拟环境里面安装Kivy的稳定版了：

```
pip install kivy
```

如果要用开发版本的Kivy，就换成下面的命令来安装：

```
pip install git+https://github.com/kivy/kivy.git@master
```

古老的PyGame相关的依赖包

Ubuntu 系统

首先还是要安装一大堆必要的系统组件了：

```
sudo apt-get install -y \
python-pip \
build-essential \
mercurial \
git \
python \
python-dev \
ffmpeg \
libsdl-image1.2-dev \
libsdl-mixer1.2-dev \
libsdl-ttf2.0-dev \
libsmpeg-dev \
libsdl1.2-dev \
libportmidi-dev \
libswscale-dev \
libavformat-dev \
libavcodec-dev \
zlib1g-dev
```

Fedora 系统

Fedora 系统就要用yum来实现组件安装：

```
sudo yum install \
make \
mercurial \
automake \
gcc \
gcc-c++ \
SDL_ttf-devel \
SDL_mixer-devel \
khrplatform-devel \
mesa-libGLES \
mesa-libGLES-devel \
gstreamer-plugins-good \
gstreamer \
gstreamer-python \
mtdev-devel \
python-devel \
python-pip
```

OpenSuse 系统

这里示范的是用zypper作为包管理器：

```
sudo zypper install \
python-distutils-extra \
python-gstreamer-0_10 \
python-enchant \
gstreamer-0_10-plugins-good \
python-devel \
Mesa-devel \
python-pip

sudo zypper install -t pattern devel_C_C++
```

然后又到了在虚拟环境中安装Kivy的时候了

首先要确保Pip, Virtualenv 和 Setuptools 这几个包都更新到最新：

```
sudo pip install --upgrade pip virtualenv setuptools
```

然后创建一个名为"kivyinstall"的新的虚拟环境，这时候你有两种选择：

- 第一种是用系统自带的默认Python解释器：

```
virtualenv --no-site-packages kivyinstall
```

- 第二种是设定一个指定位置的Python解释器，这里举例就假设要用的解释器路径在/usr/bin/python2.7

```
virtualenv --no-site-packages -p /usr/bin/python2.7 kivyinstall
```

建立好虚拟环境之后，就是进入里面了：

```
. kivyinstall/bin/activate
```

千万要注意，这里要安装numpy以及正确的Cython版本：

```
pip install numpy
pip install Cython==0.23
```

这里注意，如果你不想用sdl2，而想要用pygame，你可以通过export USE_SDL2=0来强制使用pygame。这样一来，Kivy在安装过程中找不到sdl2的链接，就会自动设置这个值为0，然后尝试用pygame来构建。

```
pip install hg+http://bitbucket.org/pygame/pygame
```

接下来就可以在这个虚拟环境里面安装Kivy的稳定版了：

```
pip install kivy
```

如果要用开发版本的Kivy，就换成下面的命令来安装：

```
pip install git+https://github.com/kivy/kivy.git@master
```

虚拟环境中安装额外的包

在该虚拟环境中安装开发版的buildozer：

```
pip install git+https://github.com/kivy/buildozer.git@master
```

安装开发版plyer

```
pip install git+https://github.com/kivy/plyer.git@master
```

其他的两个可能用到的包：

```
pip install -U pygments docutils
```

从命令行中启动

Kivy官方提供的样例代码中有一些是可以在安装配置好Kivy环境后立即就能运行的，这些例子就集成在Kivy包之内。所以，如果你要尝试这些样例，你得实现确定好easy_install把你当前在用的Kivy安装到了哪里：

```
python -c "import pkg_resources; print(pkg_resources.resource_filename('kivy', '../share/kivy-examples'))"
```

然后估计你会得到一个路径了，类似下面这样：（译者注：这个路径是根据上面那个命令来输出的，每个人不同配置都产生不同结果，千万别无脑复制哦！）

```
/usr/local/lib/python2.6/dist-packages/Kivy-1.0.4_beta-py2.6-linux-x86_64.egg/share/kivy-examples/
```

然后你知道位置了，就进入这个路径，然后运行一下样例吧。 比如你可以尝试一下触控追踪的样例touchtracer：

```
cd <path to kivy-examples #把这里替换成你自己的kivy-examples目录
cd demo/touchtracer
python main.py
```

这还有一个图片示意程序pictures：

```
cd <path to kivy-examples #把尖括号内容替换成你自己的kivy-examples目录
cd demo/pictures
python main.py
```

If you are familiar with Unix and symbolic links, you can create a link directly in your home directory for easier access. For example:

如果你对Unix和符号链接比较熟悉，你可以把这个目录在你的home目录里面创建一个链接，这样以后访问更方便，举例如下：

1 通过上面演示过的命令获取样例代码所在位置；

2 把获取的路径补齐到下列命令中，然后粘贴到终端中：

```
ln -s <path to kivy-examples ~/#把尖括号内容替换成你自己的kivy-examples目录
cd ~/kivy-examples
```

3 接下来你就可以用如下这种特别简单的方式来访问样例代码了：

```
cd ~/kivy-examples
```

如果你想更省事，把Kivy程序当做常规脚本来运行（比如输入./main.py），或者双击来运行，你就需要创建一个正确的Python链接。例如下面这样：

```
sudo ln -s /usr/bin/python2.7 /usr/bin/kivy
```

或者如果你想要在某个虚拟环境中运行Kivy，那给对应该环境的Python做个链接就行了：

```
sudo ln -s /home/your_username/Envs/kivy/bin/python2.7 /usr/bin/kivy
```

还没完，接下来你还要在每一个main.py的开头添加如下内容作为第一行：

```
#!/usr/bin/kivy
```

特别注意

一定要小心哈，Windows系统下的Python保存的文件结尾类型很可能是(CR-LF)，Linux系统不会忽略掉其中的<CR，并且依然当做文件名字的一部分来读取。这就会导致很多乱七八糟的出错信息，所以记得先确定把文件转换成Unix风格的结尾。

设备权限

当你启动app的时候，Kivy会用到Mtdev来搜索是否有可用的多指触摸设备，如果找到就拿来用作输入。然而这类设备的使用权通常都被严格限制到了特定的用户或者用户组。

如果你的用户没有这些权限，Kivy就会记下一个错误，然后给出一个与这些设备相关的警告，大概如下所示：

```
Permission denied:'/dev/input/eventX'
```

所以你要使用这些信息，就必须赋予当前用户或者用户组所必要的权限。可以通过如下命令实现：

```
sudo chmod u+r /dev/input/eventX
```

上面这个是给当前用户赋予权限，如果要给当前用户组权限，可以用下面这个命令实现：

```
sudo chmod g+r /dev/input/eventX
```

这个授权是非永久性的，这次授权后可用，以后又要重新授权才能用。所以有个更好的永久解决方案，就是把当前用户添加到有权限的用户组中。例如，在Ubuntu系统里面，你可以把这个用户添加到input这个用户组：

```
sudo adduser $USER input
```

特别注意

修改完用户权限之后，你要注销然后再登录才能使用这些权限。

Mac系统安装Kivy指南

使用官方提供的Kivy.app

特别注意

官方提供的Kivy.app仅仅适用于OS X 10.7以及更新版本的系统（都是64位的）。对10.7之前的系统以及10.7的32位系统的用户，你就只能自己手动安装各种组件了。建议通过 homebrew 来安装。

对OS X 10.7 64位版本以及更新版本系统，Kivy官方提供了一个Kivy.app的包，里面集成好了所有需要的依赖包。可以从这个链接来下载压缩包，解压缩之后就能发现一个名为**Kivy.app**的应用文件。

要怎么安装呢？具体思路如下：

1 从官网的这个链接下载压缩包，其中Kivy2.7z用的是Python 2，而Kivy3.7z用的是Python 3。

2 使用解压缩工具把压缩包进行解压，可以试试Keka这个应用。

3 把解压缩出来的Kivy2.app或者Kivy3.app这两个文件当中选择一个，重命名成Kivy.app，复制到应用程序目录/Applications 下，这个过程可以在终端中通过下面的命令来实现

```
sudo mv Kivy2.app /Applications/Kivy.app
```

4 然后是创建一个名为kivy的系统链接，以便于方便访问kivy环境来启动app：

```
ln -s /Applications/Kivy.app/Contents/Resources/script /usr/local/bin/kivy
```

5 样例代码以及所有常规的Kivy工具都可以在/Applications/Kivy.app/Contents/Resources/kivy这个目录里面找到了。

6 译者注：你完全可以同时拥有Kivy2.app和Kivy3.app，可以不重命名他们，而直接把这两个都复制到/Applications/下，然后用如下方式分别创建名为kivy2和kivy3的链接（这样以后你就可以通过kivy2来使用Python2版本的Kivy，而用kivy3来使用Python3版本的Kivy了。）：

```
ln -s /Applications/Kivy2.app/Contents/Resources/script /usr/local/bin/kivy2
ln -s /Applications/Kivy3.app/Contents/Resources/script /usr/local/bin/kivy3
```

现在你就可以在终端中用Kivy脚本文件来启动Kivy的app了，也可以把你的main.py直接拽到终端中就能运行了。

安装模块

OS X上的Kivy使用自己集成的一个python环境，只在你用kivy命令的时候才被激活。所以要在这里安装模块，要在pip命令前面加上kivy -m的前缀，如下所示（记得把替换成你要安装的模块名）：

```
kivy -m pip install <modulename>
```

这些模块安装到哪里了呢？

安装位置在Kivy.app目录内的venv目录下：

```
Kivy.app/Contents/Resources/venv/
```

如果你安装一个二进制的模块，例如kivy-garden，这些二进制文件就只能在venv一级以上的目录使用：

```
kivy -m pip install kivy-garden
```

上面这个命令安装的garden的链接库文件，只有通过如下命令激活这个虚拟环境了才能使用：

```
source /Applications/Kivy.app/Contents/Resources/venv/bin/activate
garden install mapview
deactivate
```

二进制文件安装

直接复制到/Applications/Kivy.app/Contents/Resources/venv/bin/这个目录就行了。

安装其他框架

Kivy.app自带了SDL2和Gstreamer这两个框架。要增加其他的框架让Kivy使用，可以按照如下思路实现：

```
git clone http://github.com/tito/osxrelocator
export PYTHONPATH=~/path/to/osxrelocator
cd /Applications/Kivy.app
python -m osxrelocator -r . /Library/Frameworks/<Framework_name>.framework/ \
@executable_path/../Frameworks/<Framework_name>.framework/
```

一定要记得把替换成你需要的框架名。osxrelocator这个工具是用来改变框架中的链接库目录，这样就可以让这些框架可以在Kivy.app中使用了。

启动任意一个**Kivy**应用

要运行Kivy应用，只要把源码拖拽到Kivy.app图标上，就可以了。样例代码目录中的任何Python文件都可以拿来试试。

从命令行启动

如果要在命令行中运行Kivy，把Kivy.app复制到应用目录后，双击Make Symlinks script 这个脚本文件，就可以了。要测试是否成功，可以按照如下方式：

1 打开终端，输入：

```
kivy
```

你就应该能得到一个Python解释器环境了。

2 然后在这个Python解释器内输入如下代码：

```
import kivy
```

如果什么反应都没有，没有出错，那就说明搞定了。ut errors, it worked.

3 经过上面的验证，说明配置成功了。这样在命令行终端中运行Kivy应用就很简单了，只是执行一下脚本就可以，如下所示：

```
kivy yourapplication.py
```

使用HomeBrew安装Kivy

使用HomeBrew和Pip也可以安装Kivy，具体步骤如下所示：

1 首先要先安装homebrew，然后安装必备组件：

```
brew install sdl2 sdl2_image sdl2_ttf sdl2_mixer gstreamer
```

2 然后是通过pip安装cython 0.23和kivy（一定要注意，要确保设置环境变量USE_OSX_FRAMEWORKS=0）：

```
pip install -I Cython==0.23
USE_OSX_FRAMEWORKS=0 pip install kivy
```

3 如果不想安装稳定版而想使用开发版，第二步就要改用如下命令了：

```
USE_OSX_FRAMEWORKS=0 pip install https://github.com/kivy/kivy/archive/master.zip
```

使用MacPorts和Pip来进行安装

特别注意

如果你希望自己的Kivy应用能够支持视频播放，就得手动安装gstreamer。可以供过MacPorts来安装py-gst-python port. 用MacPorts和Pip安装Kivy的过程如下：

1 安装Macports

2 安装Python 3.4并且设定成默认的：

```
port install python34
port select --set python python34
```

3 然后安装Pip并设置为默认：

```
port install pip-34
port select --set pip pip-34
```

4 使用Macports安装必备组件：

```
port install libsdl2 libsdl2_image libsdl2_ttf libsdl2_mixer
```

5 使用Pip安装cython 0.23和kivy（一定要注意，要确保设置环境变量USE_OSX_FRAMEWORKS=0）：

```
pip install -I Cython==0.23
USE_OSX_FRAMEWORKS=0 pip install kivy
```

6 如果不想安装稳定版而想使用开发版，第二步就要改用如下命令了：

```
USE_OSX_FRAMEWORKS=0 pip install https://github.com/kivy/kivy/archive/master.zip
```

特别注意

如果你在Mac系统下使用Kivy-Designer的时候遇到如下错误：

```
[WARNING           ] stderr:     from designer.app import DesignerApp
[WARNING           ] stderr:   File "/Users/cycleuser/kivy-designer/designer/app.py",
line 27, in <module>
[WARNING           ] stderr:     from kivy.garden.filebrowser import FileBrowser
[WARNING           ] stderr: ImportError: No module named filebrowser
```

那么不要犹豫，肯定是Garden的安装位置你没有调整，你需要参考我的这篇文章来解决这个问题。

Title: Kivy IDE Date: 2017-02-19 Category: Kivy Tags: Python,Kivy

在各种流行的 Python IDE 中配置 Kivy 集成开发环境

- 英文原文

在 Windows 系统上配置 PyCharm 使用 Kivy

从 1.9.1 开始， Kivy 就可以安装到你系统中已有的 Python 解释器中，所以在 Windows 系统上面的安装非常简单直接。

- 1 在 Windows 系统上安装 Kivy；可以参考英文原版安装指南或者我博客里的安装指南,或者参考我的知乎专栏。
- 2 然后就在 PyCharm 里面建立或者打开你的项目就可以了。

理论上就这么简单，两步就搞定了。如果你有多个 Python 解释器，那就需要选择安装了 Kivy 的那个才行。具体步骤是在 PyCharm 里面，按照如下流程操作：

- 1 File 文件 -> Settings 设置 -> Project 项目 -> Project Interpreter 项目解释器
- 2 取消掉当前的项目解释器，选择你要用的安装了 Kivy 的 Python 解释器就好了。

这就完毕了，接下来就可以用 Kivy 来玩耍了。

在 macOS 系统上配置 PyCharm 使用 Kivy

Kivy 开发团队推荐 macOS 用户使用 homebrew 来安装 Kivy。这很简单，在 Mac 系统里面打开 Terminal 终端就可以了。下面的代码输入或者复制粘贴到终端提示符后面就可以了。 要注意这些代码特别长，所以可能因为文字换行等等导致出现多行的情况，这时候一定要注意要复制完整。 除了用 brew 和 pip 来安装 Kivy 的方法之外，还可以参考其他的详细安装指南，比如我博客里的安装指南或者我的知乎专栏。

- 1 安装 Homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/mast
er/install)"
```

- 2 通过 Brew 来安装 Python 3

```
brew install python3
```

或者 Python 2

```
brew install python
```

- 3 安装 Kivy （参考我博客里的安装指南或者我的知乎专栏）。这里一定要注意，下面的命令使用了 pip3 ，意思是说安装到了 Python3 里面，如果你要用 Python2 ，那就把下面命令中的 pip3 替换成 pip：

```
brew install hg sdl sdl2 sdl2_image sdl2_ttf sdl2_mixer gstreamer
pip3 install --upgrade pip
pip3 install cython
USE_OSX_FRAMEWORKS=0 pip3 install kivy
```

PyCharm 应该能选择你默认的 Python 解释器，然后（估计差不多）就是你刚刚把 Kivy 座位一个模块来安装好的那个解释器。

可以通过下面的路径来对已有模块进行检查： File 文件 -> Settings 设置 -> Project 项目 -> Project Interpreter 项目解释器

Kivy 语言自动补齐以及代码高亮

ID 为 Xuton 的一位好心朋友开发了一个扩展文件，可以提供 Kivy 语言的自动补齐和代码高亮，安装方法如下：

- 下载这个文件；
- 打开 Pycharm 的主菜单，点击 File 文件 -> Import 导入 (或者 Import Settings 导入设置) ；
- 选择刚刚下载的那个 jar 文件，然后 PyCharm 会弹出一个对话框让你来确认，点击 OK 就行了。
- 重新启动 PyCharm ，看看是不是有效果了？

更多其他 IDE

这些我就不翻译了，大家可以去看看。

- How to use Python Tools for Visual Studio to develop and run Kivy Applications
- kivy-eclipse-and-pydev-also-pypy
- How to Set up Wing IDE for Kivy on Windows

- Kv language definition file for gedit

Title: Kivy Basics Date: 2017-01-22 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：基础知识

原文地址。

Kivy环境安装搭建

Kivy要依赖很多Python包，比如 pygame, gstreamer, PIL, Cairo 等等还有好多。这些包并非都是必需的，要根据你的运行平台来看具体情况，有时候缺那么一两个包就可能导致安装失败，或者运行过程中出错等等，这就挺痛苦的。所以Kivy官方针对Windows和MacOS X提供了集成好关键部件的压缩包，解压缩之后直接就能用。具体的安装过程可以参考下面链接中的中文安装指南：

- Kivy中文安装指南

如果你非要自己从零开始安装，那最起码要确保安装有Cython和Pygame。这两个包可以通过pip来安装，如下所示：

```
pip install cython
pip install hg+http://bitbucket.org/pygame/pygame
pip install kivy
```

Kivy的开发版本也可以通过git来安装：

```
git clone https://github.com/kivy/kivy
make
```

创建一个应用

创建一个Kivy应用挺简单的，大概步骤如下：

- 基于App类创建一个子类；
- 把build()方法实现为返回一个控件实例(这个控件的实例也就是你整个应用的根控件)。
- 创建一个这个类的实例，然后调用run()方法。

下面的代码就是上述思路的最小化实现：

```
import kivy
kivy.require('1.0.6') # 注意要把这个版本号改变成你现有的Kivy版本号!

from kivy.app import App # 译者注：这里就是从kivy.app包里面导入App类
from kivy.uix.label import Label # 译者注：这里是从kivy.uix.label包中导入Label控件，这里都注意开头字母要大写

class MyApp(App):

    def build(self): # 译者注：这里是实现build()方法
        return Label(text='Hello world') # 译者注：在这个方法里面使用了Label控件

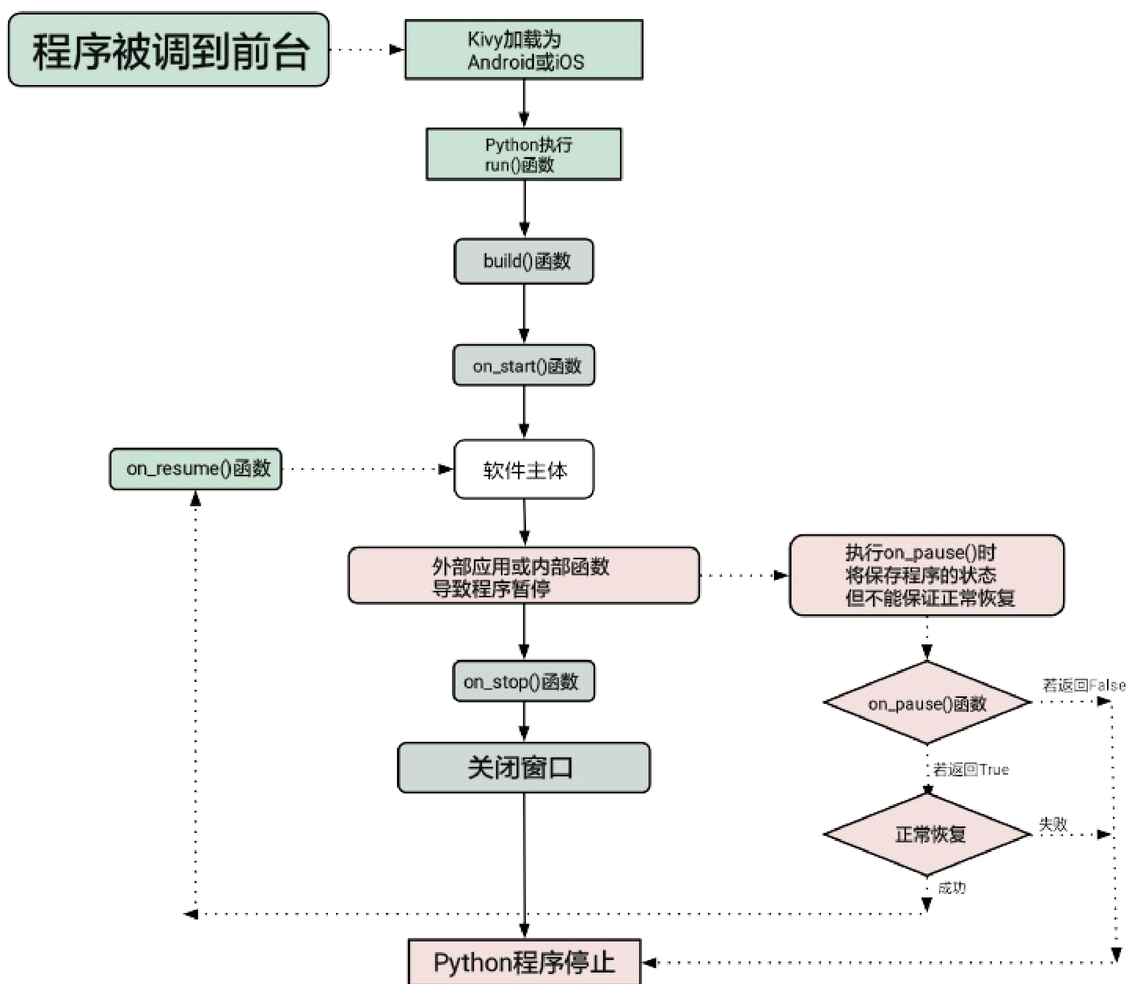
if __name__ == '__main__':
    MyApp().run() # 译者注：这里就是运行了。

'''
译者注：这一段的额外添加的备注是给萌新的.
就是要告诉萌新们，一定要每一句每一个函数每一个变量甚至每一个符号，都要读得懂！！！
如果是半懂不懂的状态，一定得学透了，要不然以后早晚得补课.
这时候又让我想起了结构化学。
总之更详细的内容后面会有，大家加油。
'''
```

把上面的代码以文本形式复制到一个文本文件中，保存成py文件，例如main.py，然后运行，就行了。

Kivy应用的生命周期

跟学习开发Android应用的时候类似，咱们首先也是要了解一下Kivy应用的生命周期（生命周期图由小涂汉化）：



如上图所示，不论什么用途和目的，咱们应用的入口都是这个run()方法，在本文的样例代码中，就是“MyApp().run()”。

下面就一行一行开始详细解释了：

```
from kivy.app import App
```

为什么要导入这个App类呢？因为咱们自定义的这个App要继承这个类。这个类的位置在kivy安装目录下的kivy目录下的app.py文件中。

特别注意

如果你想要深入挖掘一下，去了解这个Kivy的App类到底是怎么个内容，你可以打开这个app.py文件，亲自来看看。Kivy作者特别鼓励大家去阅读源码。Kivy基于Python，用Sphinx编写的文档，所以每个类的文档都在对应的文件内。

然后咱们回过头来，继续看本文这次的代码的第二行：

```
from kivy.uix.label import Label
```

这里一定要特别注意各种包和类的导入。"kivy.uix"这个包的作用是容纳用户界面元素，比如各种输出布局和控件。

接下来看到这一行：

```
class MyApp(App):
```

这一行定义了咱们这次的Kivy应用的基类。如果你要做修改的话，把MyApp改成你要设定的应用名字就可以了。

接着往下看：

```
def build(self):
```

在上面的生命周期图中加粗强调的部分表明，build函数所处的是要进行初始化和返回根控件的位置。根控件返回的操作在下面这一行中实现：

```
return Label(text='Hello world')
```

这里我们用文本‘Hello World’对Label这一控件进行了初始化，并且返回了其实例。这个Label就是咱们这个应用的根控件了。

特别注意

Python是用缩进来区别代码块的，所以一定要注意上面代码的缩进和层次，尤其是函数定义那部分。

然后咱们继续，到了真正让应用开始运行的这部分了：

```
if __name__ == '__main__':
    MyApp().run()
```

这里对MyApp这个类进行了初始化，然后调用了这个类的run()方法。这样就初始化并启动了我们的Kivy应用了。

运行应用

接下来就是要在不同操作系统平台上来运行咱们刚刚写好的应用了：

To run the application, follow the instructions for your operating system:

- Linux 终端中以如下方式运行：

```
$ python main.py
```

- Windows 可以在CMD中以如下方式运行：

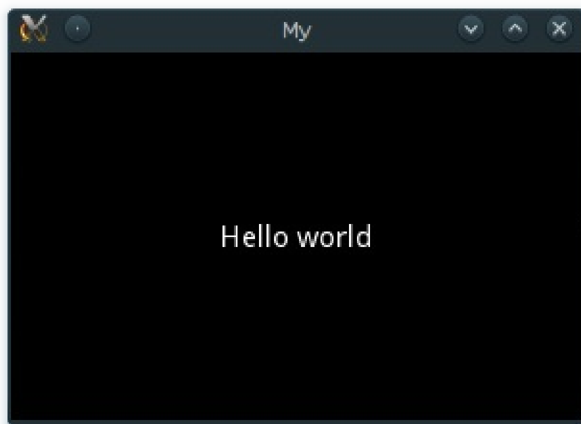
```
$ python main.py #用系统Python运行
C:\appdir>kivy.bat main.py #用kivy.bat来运行，注意这里要设定好正确的路径
```

- Mac OS X 跟Linux差不多，也在终端中运行，不过是要用Kivy官方提供的集成解释器：

```
$ kivy main.py
```

- Android 下面要运行还需要一些复杂的文件，所以等以后深入了之后再给讲解这部分了。

这个应用运行之后的具体效果就是下面图片所示这样，会打开一个窗口，然后展示出一个Label，上面写着文本'Hello World'，这个Label会覆盖该窗口的全部区域。就这样了。



修改定制这个应用

接下来咱们扩展一下这个应用的功能，增加一个用户名/密码输入的页面吧。

```
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.textinput import TextInput

class LoginScreen(GridLayout):

    def __init__(self, **kwargs):
        super(LoginScreen, self).__init__(**kwargs)
        self.cols = 2
        self.add_widget(Label(text='User Name'))
        self.username = TextInput(multiline=False)
        self.add_widget(self.username)
        self.add_widget(Label(text='password'))
        self.password = TextInput(password=True, multiline=False)
        self.add_widget(self.password)

class MyApp(App):

    def build(self):
        return LoginScreen()

if __name__ == '__main__':
    MyApp().run()
```

在下面这行代码中，我们导入了一种名为Gridlayout的布局：

```
from kivy.uix.gridlayout import GridLayout
```

这个类被我们用作基类来制作根控件LoginScreen，在如下代码中进行了定义：

```
class LoginScreen(GridLayout):
```

如下代码中，我们在LoginScreen类中重新定义了初始化方法**init**()，这样来增加一些控件，并且定义了这些控件的行为：

```
def __init__(self, **kwargs):
    super(LoginScreen, self).__init__(**kwargs)
```

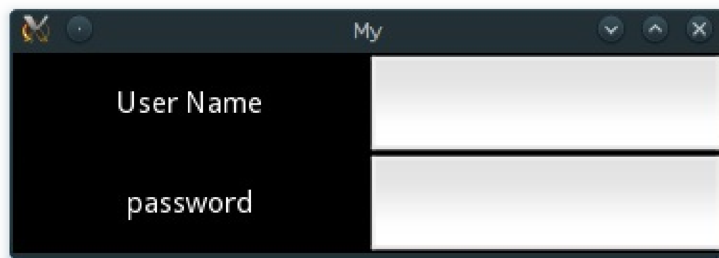
一定要注意这里要加super，才能把现有的新初始化方法覆盖掉继承来的旧初始化方法。另外也要注意，这里调用super的时候没有省略掉**kwargs，这是一种好习惯。

然后继续往下看：

```
self.cols = 2
self.add_widget(Label(text='User Name'))
self.username = TextInput(multiline=False)
self.add_widget(self.username)
self.add_widget(Label(text='password'))
self.password = TextInput(password=True, multiline=False)
self.add_widget(self.password)
```

上面的代码中，我们让GridLayout来管理子控件，把子控件设置为两栏，然后加上用户名和密码的Label字符显示控件和TextInput字符输入控件。

运行上面的代码，得到的窗口效果大概如下图：



尝试着重新缩放一下窗口大小，你会发现上面的控件会相对整个窗口的尺寸而自行调整大小，并不需要人为去操作了。这是因为这些控件都使用了默认的尺寸。

上面这个代码虽然有输入框，但是并没有提供用户输入的支持和处理，所以并不能进行用户名/密码验证，也没有任何其他用处。后续的练习中咱们再来深入去探讨这些功能，并且还会讲一讲空间的尺寸和位置等话题。

Title: Kivy Environment Date: 2017-02-5 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：环境变量

英文原文

Kivy的初始化和很多行为都可以通过各种环境变量来控制。

例如，若要严格设定用PIL进行文本渲染，可以通过如下方式来实现：

```
$ KIVY_TEXT=pil python main.py
```

(译者注：PIL，Python Imaging Library, Python 下常用的绘图库)

所有的这些环境变量的修改设定都需要在导入Kivy之前进行，具体如下所示：

```
import os
os.environ['KIVY_TEXT'] = 'pil'
import kivy
```

路径控制

- 从Kivy1.0.7版本开始提供

You can control the default directories where config files, modules and kivy data are located.

Kivy的配置文件、模块以及数据存储的默认目录，都可手动设定所在位置。

> KIVY_DATA_DIR

这个是Kivy的数据目录，默认值为/data 。

> KIVY_MODULES_DIR

这个是Kivy的模块目录，默认值为/modules 。

> KIVY_HOME

这个是Kivy的HOME目录，该目录是用来存放本地配置文件的，必须是一个可以写入的位置。对应不同系统也有不同位置：

```
* Desktop: /.kivy
* Android: /.kivy
* iOS: /Documents/.kivy
```

- 从Kivy1.9.0版本开始提供

> KIVY_SDL2_PATH

这个变量若设定了，编译Kivy的时候就会使用该位置的SDL2库文件，而不再使用系统的库文件。在环境变量PATH的开头部位就要设定好这个变量，这样在运行一个Kivy应用的时候才能也使用相同的SDL2库文件。

- 从Kivy1.9.0版本开始提供

特别注意

刚刚这个SDL2路径是用来编译Kivy的。运行程序的话就用不着了。

配置文件

> KIVY_USE_DEFAULTCONFIG

若设定了此环境变量，Kivy会读取制定的配置文件。

> KIVY_NO_CONFIG

若设定了此环境变量，Kivy将不会读取也不会写入任何配置文件。也适用于用户配置文件夹的位置。（译者注：这句话我还没弄明白，因为没有这样尝试。）

> KIVY_NO_FILELOG

若设定了此环境变量，日志将不再输出到文件内。

> KIVY_NO_CONSOLELOG

若设定了此环境变量，日志将不再输出到控制台。

> KIVY_NO_ARGS

若设定了此环境变量，命令行传递的参数将不会被Kivy解析和使用。也就是说，可以不用 – 定义符，就能随便创建一个使用自己参数的脚本或者应用：

```
import os
os.environ["KIVY_NO_ARGS"] = "1"
import kivy
```

- 从Kivy1.9.0版本开始提供

限定Kivy.core核心，使用特定版本

Kivy.core会尝试使用所在平台的最优实现。如果要测试或者定制安装，你可能要把选择器设定为某个特定版本的kivy.core。

> KIVY_WINDOW

这一变量是用来设定如何创建窗口，可用值：sdl2, pygame, x11, egl_rpi

> KIVY_TEXT

这一变量是用来设定如何渲染文本，可用值：sdl2, pil, pygame, sdlttf

> KIVY_VIDEO

这一变量是用来设定如何渲染视频，可用值：pygst, gstplayer, pyglet, ffpyplayer, ffmpeg, gi, null

> KIVY_AUDIO

这一变量是用来设定如何播放声音，可用值：sdl2, gstplayer, pygst, ffpyplayer, pygame, gi, avplayer

> KIVY_IMAGE

这一变量是用来设定如何读取图像，可用值：sdl2, pil, pygame, imageio, tex, dds, gif

> KIVY_CAMERA

这一变量是用来设定如何读取摄像头，可用值：videocapture, avfoundation, pygst, opencv

> KIVY_SPELLING

这一变量是用来设定拼写，可用值： enchant, osxappkit

> KIVY_CLIPBOARD

这一变量是用来设定剪切板管理组件，可用值：sdl2, pygame, dummy, android

设置单位

> KIVY_DPI

这个是用来设定Metrics.dpi的dpi值的。

- 从Kivy1.4.0版本开始提供

> KIVY_METRICS_DENSITY

这个是用来设定Metrics.density，像素密度。

- 从Kivy1.5.0版本开始提供

> KIVY_METRICS_FONTSCALE

这个是用来设定Metrics.fontscale，字体大小。

- 从Kivy1.5.0版本开始提供

图形输出

> KIVY_GL_BACKEND

此变量用于设定使用的OpenGL后端，更多细节参考cgl.

> KIVY_GL_DEBUG

此变量用于设定是否对OpenGL调用进行日志记录，更多细节参考cgl.

> KIVY_GRAPHICS

此变量用于设定是否使用OpenGL ES2，更多细节参考cgl.

> KIVY_GLES_LIMITS

此变量用于设定是否强制设定GLES2（默认值为启用，设置为1）。如果设定为false，Kivy将不再兼容GLES2。（译者注：这部分我不懂，就直接生硬翻译了原文，建议大家参考一下原文去理解。）如果设置为true，可能有下表中所示的潜在的不兼容情况：

Mesh indices	If true, the number of indices in a mesh is limited to 65535
Texture blit	When blitting to a texture, the data (color and buffer) format must be the same format as the one used at the texture creation. On desktop, the conversion of different color is correctly handled by the driver, while on Android, most of devices fail to do it. Ref: https://github.com/kivy/kivy/issues/1600

- 从Kivy1.8.1版本开始提供

> KIVY_BCM_DISPMANX_ID

此变量是针对Raspberry Pi树莓派平台的，用于设定所选择的视频输出端口。默认值为0，下面列表中是在vc_dispmanx_types.h这个头文件中存储的可供选择的变量值：

- 0: DISPMANX_ID_MAIN_LCD
- 1: DISPMANX_ID_AUX_LCD
- 2: DISPMANX_ID_HDMI
- 3: DISPMANX_ID_SDTV
- 4: DISPMANX_ID_FORCE_LCD
- 5: DISPMANX_ID_FORCE_TV

- 6: DISPMANX_ID_FORCE_OTHER

（译者注：上面0-6分别是不同的显示输出端口，相信很容易看懂，大家探索一下吧。）

Title: Kivy Configure Date: 2017-02-6 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：配置修改

英文原文

Kivy的配置文件是一个名为config.ini的文本，符合标准INI格式。

找到配置文件位置

Kivy的配置文件存放在环境变量KIVY_HOME所制定的位置：

```
KIVY_HOME>/config.ini
```

在桌面平台上，默认的位置如下：

```
HOME_DIRECTORY>/.kivy/config.ini
```

所以，假设你的用户名是“tito”，在各个操作系统下的配置文件位置则如下所示：

- Windows: `C:\Users\tito\.kivy\config.ini`
- OS X: `/Users/tito/.kivy/config.ini`
- Linux: `/home/tito/.kivy/config.ini`

（译者注：这里要注意，tito只是原文的一个示范，相当于张三李四这样，新手可别照着复制找不到，要用自己操作系统中具体的用户名。）

在Android系统中位置如下：

```
ANDROID_APP_PATH>/.kivy/config.ini
```

假如你的Kivy应用的包名称为“org.kivy.launcher”,那么该Kivy应用的配置文件位于:

```
/data/data/org.kivy.launcher/files/.kivy/config.ini
```

在iOS上Kivy的默认配置文件位于：

```
HOME_DIRECTORY>/Documents/.kivy/config.ini
```

本地配置

有时候用户或者开发者可能需要针对特定的应用来修改配置，或者对Kivy的某个组件进行测试，比如输入模块之类的。这时候就可以用如下命令创建一份新的配置文件：

```
from kivy.config import Config
Config.read(file>)
# set config
Config.write()
```

有时候本地配置只有一个.ini文件还不够用，比如说可能你要单独使用某个garden、Kivy日志或者其他什么模块，这时候就要把KIVY_HOME这个环境变量进行修改了，指定到目标位置就行：

```
import os
os.environ['KIVY_HOME'] = folder>
```

还有一种思路，就是在运行Kivy应用之前，在终端中手动修改一下这个环境变量：

1. Windows: `set KIVY_HOME=folder>`
2. Linux & OSX: `export KIVY_HOME=folder>`

在设置了KIVY_HOME之后，所指定的这个文件夹就会被当做默认的.kivy文件夹来用。

详细理解配置项

在kivy.config 模块中可以看到全部的配置项的解释。

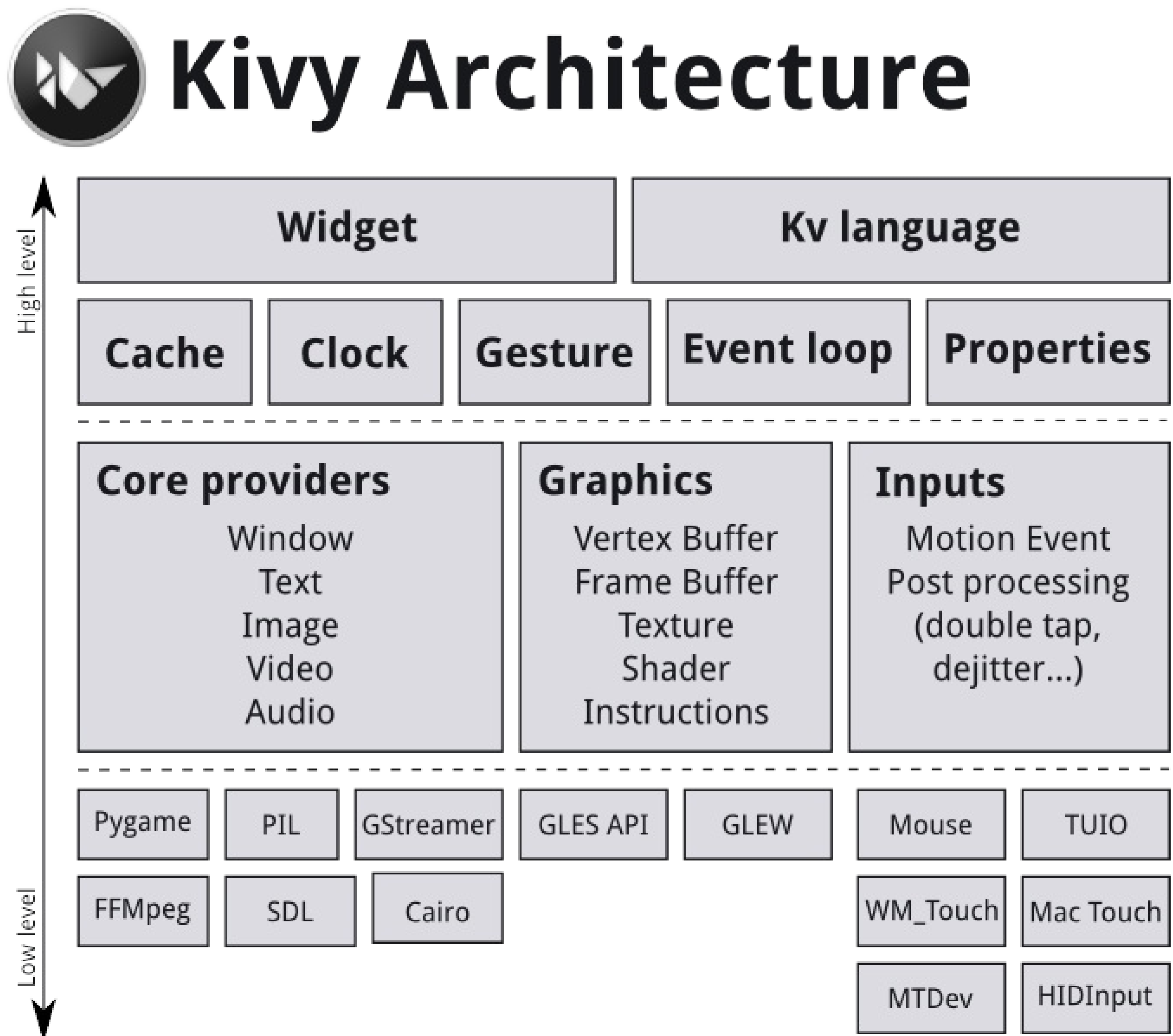
Title: Kivy Architecture Date: 2017-02-7 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：架构概览

英文原文

本章我们将从软件工程的角度，来简单介绍一下Kivy的设计。这对于理解各个部分如何配合工作会有帮助。如果你只关注代码，可能有时候会遇到这样一种情况，就是你已经有了一个初步的想法了，但具体怎么去实现可能还是一头雾水，所以本章就针对这种情况，来更深入地讲解一下Kivy的一些基本思想。你也可以先跳过这一章，等以后再翻回来看，不过我们建议开发者还是先看一下这些内容比较好，起码可以快速略读一下有个印象。

Kivy包含的若干个模块，我们将对这些模块一一进行简要说明。下面这幅图是Kivy整个架构的概括图示：



核心模块和输入模块

对理解Kivy的设计内涵，模块化和抽象化的思想是至关重要的。我们试图把各种基本的任务进行抽象，比如打开窗口、显示图像和文本、播放音频、从摄像头获取图像、拼写校正等等。我们将这些部分称为核心任务。这样也使得API接口用起来比较简单，扩展起来也容易。更重要的是，这种思路可以让Kivy应用在运行的时候，使用各个运行平台所提供的对应功能的API接口。例如，在苹果的MacOS操作系统、Linux操作系统和Windows操作系统中，就都有各自不同的原生API接口提供各种核心功能。所以就有一部分代码，调用这些不同接口中的某一个，一方面与操作系统进行通信，另一方面与Kivy进行交互，起到中间人的角色，我们称之为核心模块。针对不同的操作系统平台要使用各自对应的核心模块，这样的好处是达到一种均衡状态，既能够充分利用操作系统提供的功能，又能尽量提高开发效率。（译者注：我的理解是这样大家平时不用针对不同操作系统去学习和使用各自的API，而只要专心使用Kivy的核心模块进行调用就行了。）这也允许用户来自由选择，使用Kivy提供的核心模块，或者直接使用各个操作系统的API接口。此外，由于使用了各个平台所提供的链接库文件，我们大大减小了Kivy发型版本的体积，也使得打包发布更加容易。这有助于将Kivy应用移植到其他平台。比如Android平台上的Kivy应用就体现了这一特性的好处了。

在输入模块这部分，我们也遵循了同样的思路。输入模块，是一段代码，用于针对各种输入设备提供支持，比如苹果公司的Trackpad触摸板，TUIO多点触摸设备，或者是鼠标模拟器等等。如果你需要对某一种新的输入设备添加支持，只需要提供一个新的类，用这个类来读取输入设备的数据，然后传递给Kivy基本事件，就可以了。

图形接口

Kivy的图形接口是对OpenGL的抽象。在最底层，Kivy使用OpenGL的命令来进行硬件加速的图形绘制。不过写OpenGL的代码可还是挺复杂的，新手就更难以迅速掌握了。所以我们就提供了一系列的图形接口，利用这些接口可以很简单地进行图形绘制，这些接口中用到了例如画布Canvas、矩形Rectangle等几何概念，比OpenGL里面简单不少。

Kivy自带的所有控件，都使用了这个图形接口；出于性能的考虑，此图形接口是用C语言来实现的。

这个图形接口的另一个好处是可以对你代码中的绘图指令进行自动优化。这个很有用，尤其是在你对OpenGL的优化不太熟悉的情况下。这能让你的绘图代码更高效。

当然了，你也可以坚持使用原生的OpenGL命令。目前Kivy在所有操作系统平台上用的都是是OpenGL 2.0 ES (GLES2)，所以如果你希望保持跨平台的兼容性，我们建议你只是用GLES2兼容的函数。

核心模块

核心模块也就是kivy.core，这个包里面提供了常用的各种功能，比如：

- Clock

  时钟类，可以用于安排计时器事件。同时支持一次性计时和周期性计时。

- Cache

  If you need to cache something that you use often, you can use our class for that instead of writing your own.

  缓存类，如果有一些经常用到的数据需要缓存，就可以用这个类，而不用自己写了。

- Gesture Detection

  手势识别，这个可以用来识别各种划动行为，比如画个圆圈或者方块之类的。可以训练来识别你自己设计的图形。

- Kivy Language

  Kivy语言，这个是用来简洁高效地描述Kivy应用的用户界面的。

- Properties

  这里这些属性和Python语言中的属性不同。这里是我们专门写的一些类，通过用户界面描述来连接控件代码。

UIX（控件和布局）

UIX用户界面模块，包含了常用的各种控件和布局，可以通过复用来快速构建用户界面。

- Widgets 控件

  控件是各种用户界面元素，可以添加到程序中来提供各种功能。有的可见，有的不可见。文件浏览器，按钮、滑动页、列表等等，这都属于控件。控件接收动作事件。

- Layouts 布局

  布局是控件的排列方式。当然，你也可以自己自定义控件的位置，不过从我们提供的预设布局中选择一个来使用，会更方便很多。网格布局、箱式布局等等，都是布局了。你还可以试试复杂的多层网状布局。

模块化

如果你用某一种现代的网络浏览器，并且通过一些附加组件对其进行定制，那么你应该就理解了我们提供的各种模块类的基本思想了。各种模块可以用来向Kivy程序中添加功能，即便原作者没有提供的功能也可以加进去了。

例如，有一个模块就能显示当前应用的FPS（Frame Per Second，每秒帧数，即帧率），然后还能统计一段时间的FPS变化。

你可以自己写各种模块添加到应用中。

输入事件（Touches）

Kivy抽象了各种输入类型和输入设备，比如触控，鼠标按键，多点触摸等等。这些输入类型有一个共同点，就是都可以把各种输入事件映射成屏幕上对应的一种2D形态。（当然了，还有的输入设备就没法用2D形态来表示，比如通过加速度传感器来衡量设备倾斜角度等。这种情况就得另外考虑了。下面我们讨论的只是那些能用2D形态表示的输入事件，复杂的类型以后再说。）

这些输入类型，在Kivy中都用Touch()类的实例来表示。（请注意，这里可不仅仅是针对手指去触摸的那种touch，而是所有可以这样抽象表示的输入事件。这里用**Touch**只是为了方便而这么简称一下。就想象一下，这些**Touches**就是在用户界面或者显示屏上面的那些个点击行为。）Touch的实例或者对象，有三种状态。当这个Touch进入了其中的某一个状态，你的程序就会被告知此事件的发生。Touch的三种状态如下：

- Down 落下

  处于落下状态，只能有一次，就是在发生Touch事件的初始时刻。

- Move 移动

  这个状态的时间无上限。在一个Touch的生命周期中可以没有这个状态。移动状态只发生在Touch的2D平面位置发生变化的情况下。

- Up 抬起

  A touch goes up at most once, or never. In practice you will almost always receive an up event because nobody is going to hold a finger on the screen for all eternity, but it is not guaranteed. If you know the input sources your users will be using, you will know whether or not you can rely on this state being entered.

  一个Touch要么只能抬起一次，要么就不发生。而实际应用中你会经常遇到Up时间，因为没有人会一直把手指按到屏幕上，不过也有未必就绝对不会有这种情况。若事先知道用户用的输入设备，就可以确定能否完全依靠用户的输入状态。

（译者注：以手指触摸屏幕为例，只有开始接触的时候是Down手指落下这个状态，之后移动就是接下来的Move移动状态，手指抬起来的时候就是Up即抬起状态了；如果以鼠标左键点击为例，按下去左键的时候是Down，按住左键不放进行拖动就是Move，松开左键就是Up了。这段我特别解释一下，因为自己翻译的太生硬了。）

控件和事件调度

在图形化的软件开发语境下，控件这个词经常出现，一般是来描述程序中用于和用户进行交互的组件。在Kivy中，控件是用来接收各种输入事件的。所以并不一定非要在屏幕上能看得到。Kivy当中所有控件都以控件树的形式来管理，学过计算机科学中数据结构相关知识的话，就会意识到这是一种树形结构：一个控件可以有任意多个子控件，也可以没有子控件。根控件就只能有一个，处于树形结构的顶端，根控件不具有父控件，并且所有其他控件都是根控件的直接或者间接子控件（就像树根一样，所以叫根控件）。

当新的输入数据可用的时候，Kivy会针对每一个Touch发出一个事件。控件树种的根控件首先接收到这个事件。Touch的不同状态，on_touch_down, on_touch_move和on_touch_up（Down落下、Move移动和Up），会作为Touch的参数，提供给根控件，根控件会调用对应的事件Handler来作出反应。

包括根控件在内，控件树种的每个控件都可以有两种选择，处理该事件，或者将该事件传递下去。如果一个事件的Handler返回True，就意味着这个事件已经被接收并妥善处理。这个事件就也到此为止了。如果不是这样，事件的Handler会跳过此处的空间，调用父类中的对应事件的Handler实现，传递给该控件的子控件。这样的过程可以一路走到最基础的控件类Widget，在它的Touch事件Handler中，只是把Touch传递给子控件，而不进行其他的操作。

```
# This is analogous for move/up:
def on_touch_down(self, touch):
    for child in self.children[:]:
        if child.dispatch('on_touch_down', touch):
            return True
```

说起来挺麻烦，看上去挺复杂，实际上要简单得多。下一章就会讲解如何使用这种特性来快速创建应用了。

经常有一种情况，就是你可能要让一个控件只在屏幕上某个特定的区域来监听Touch事件。这时候就可以使用控件的collide_point()方法来实现此目的。只需要把Touch的位置发给该方法，然后如果此位置位于监听区域则返回True，反之返回False。默认情况下，这个方法会监听屏幕上的一个矩形区域，根据控件的中心坐标（x & y坐标系），以及空间尺寸（宽度和高度），不过你也可以用自己的类覆盖掉这一行为。

Title: Kivy Events and Properties Date: 2017-02-9 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：事件和属性

英文原文

译者前言

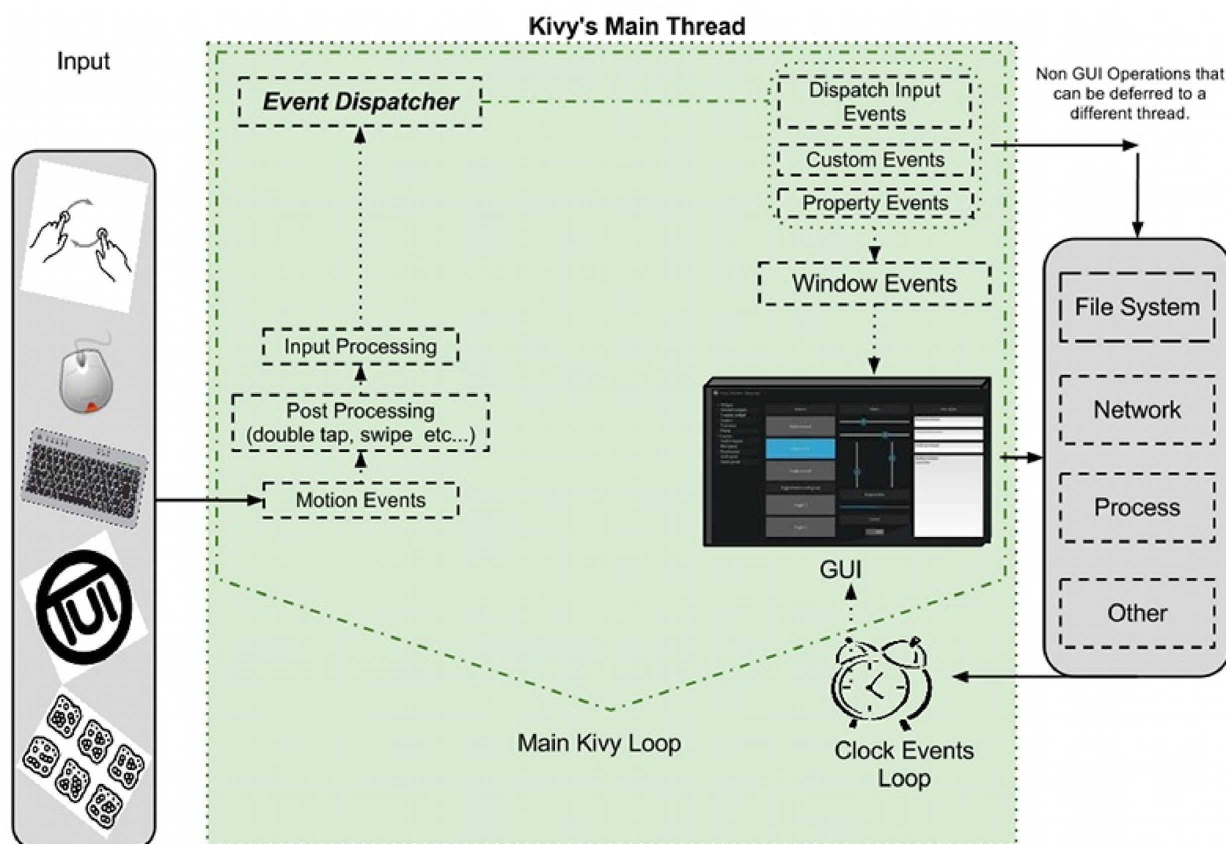
这一章节是我有史以来翻译质量的低谷，一来是我自己也是刚学，半懂不懂，二来本身语言基础各方面也薄弱，三来是笔记本坏掉了，搞个ChromeOS折腾中。

大家凑合看看，看不下去给指出来一下比较不好理解和绕的地方，以及错误的地方，我一定即时修改。

简要介绍

在Kivy开发过程中，事件是最重要的一部分了。如果之前有过GUI图形界面开发的经验的话，你可能对此习以为常了，但对新手来说，这个概念很重要。一旦你理解了事件的应用和搭配，你就会发现在Kivy开发的过程中，事件是无处不在的。有了各种事件的搭配，你就可以用Kivy来搭建你想要的各种功能了。

下面这幅图展示了Kivy框架中事件的处理过程：



事件分派器

事件分派器EventDispatcher c是Kivy框架中最重要的基类之一。通过这个类，用户可以注册各种事件，然后分发给对应的部件（一般情况下是其他的事件分派器）。控件类Widget, 动画类Animation 以及时间类Clock 都属于事件分派器。

事件分派器对象要在整个程序的循环流程的基础上来生成和处理各种事件。

主流程

上文的简介概括起来是说，Kivy有一个主循环体。这个循环体会在Kivy应用的整个生命周期中一直运行，直到退出应用的时候才结束。

在循环内部，每一步迭代都伴随有各种事件生成，这些事件可以来自用户输入、硬件传感器，或者是其他的各种来源，然后一帧一帧地渲染到屏幕上。

你写的应用程序将要指定好各种由主循环进行调用而产生的回调（callback，稍后再详细介绍相关内容）。如果一次回调花费很长时间或者根本不退出，主循环就被打破了，你的应用也就不能正常工作了。

在Kivy应用里面，一定要避免用特别长的循环、无限循环，或者休眠。下面这个代码就同时是死循环+休眠，就是一个反例了：

```
while True:
    animate_something()
    time.sleep(.10)
```

如果把上面这段代码拿去运行，那程序就会无法退出循环了，就让Kivy卡住了，什么后续步骤都不能进行了。用户就只能看到一个黑色的窗口，什么操作都没有响应。不能死循环也不能休眠，所以就得想其他办法，比如有计划地重复调用对animate_something()这样的函数。（animate_something 的意思是让某个东西动起来，作者是用来指代类似的这种需要时不时重复调用的函数。）

计划周期事件

利用schedule_interval()这个函数，你就可以每秒对某个函数或者方法进行指定次数的调用了。下面就是一个例子，在这段代码的第三行，实现了每秒钟调用my_callback(dt)函数三十次：

```
def my_callback(dt):
    print 'My callback is called', dt
event = Clock.schedule_interval(my_callback, 1 / 30.)
#译者注：这里的1/30明显就是频率的倒数了，如果是每秒钟50次就应该是1/50，以此类推了，大家可以自己修改试试看。
```

要取消之前的计划事件有多种方法。可以用cancel()，也可以用 unschedule()：

```
event.cancel()

#或者用下面这种方法

Clock.unschedule(event)
```

再有一种方法，就是在回调的时候返回False，这样这个事件就会被自动取消计划，不再重复：

```
count = 0
def my_callback(dt):
    global count
    count += 1
    if count == 10:
        print 'Last call of my callback, bye bye !'
        return False
    print 'My callback is called'
Clock.schedule_interval(my_callback, 1 / 30.)
```

计划一次性事件

使用schedule_once()函数，可以对一个函数稍后调用的效果，可以是在下一帧，也可以是在指定时间X之后：

```
def my_callback(dt):
    print 'My callback is called !'
Clock.schedule_once(my_callback, 1)
```

上面这段代码会在一秒钟后再对my_callback(dt)进行调用。schedule_once()函数的第二个变量X就是延迟调用的时间，以秒为单位。具体这个变量的用法有以下三种：

- 若X大于零，则作为时间长度的秒数，延迟X秒之后进行下一次调用
- 若X等于零，则在下一帧进行调用
- 若X为-1，调用则发生在下一帧渲染之前

假如你已经有了一个计划事件，但又想要在下一帧渲染之前计划一次调用，这种情况就适合使用-1这种用法。

这里就有了一种衍生出来的重复调用某个函数的方法，就是在函数体内放一个schedule_once()，然后在第二次调用该函数的时候，函数内的schedule_once()就会继续对本身进行调用了：

```
def my_callback(dt):
    print 'My callback is called !'
    Clock.schedule_once(my_callback, 1)
Clock.schedule_once(my_callback, 1)
```

主循环会一直按照代码的要求来保持各种计划调用的实现，但一次计划调用发生的具体时间是具有一些不确定性的。有时候其他的调用或者应用中的其他任务可能会比想象中执行得更久一些，这时候用计时的方法制定计划就不太合适了。

后面介绍的这种用内置schedule_once()来进行重复回调问题的解决方案中，在最后一次迭代结束后，下一次迭代将至少要一秒之后才能被调用。 而使用schedule_interval()这种方法就可以每秒都进行回调。

触发事件

有时候可能一个函数只需要计划在下一帧调用一次，而不允许重复调用。这时候就可以用下面这样的思路来实现：

```
#首先是用schedule_once()计划调用一次
event = Clock.schedule_once(my_callback, 0)

#然后在另外一个位置，就用unschedule()取消计划调用，这样就能避免重复调用。接下来就是再次用schedule_once()进行计划调用。
Clock.unschedule(event)
event = Clock.schedule_once(my_callback, 0)

#译者注：我这部分理解的也不够深，翻译得很生硬，我的大概理解就是这样可以精确控制调用次数，避免一次计划调用之后发生的调用次数不可控。
```

上面这种方法构建触发器可谓费时费力，因为你得经常用到unschedule，即使一个事件已经结束。此外，每次还都产生新事件。所以可以用下面这个trigger()来作为触发器：

```
trigger = Clock.create_trigger(my_callback)
# later
trigger()
```

这样你每次调用trigger()就可以了，这个触发器会对你的my_callback回调进行单次计划调用。如果之前存在计划调用了，则不重新产生计划调用。

控件事件

控件有两种默认事件：

- 属性事件：比如你的控件改变了位置或者大小，就会触发一个事件。
- 控件定义的事件：比如一个Button按钮控件被按下或者松开，也会触发一个事件。

关于控件的Touch事件的管理和传播，可以参考API文档中这部分相关内容。

创建自定义事件

要使用自定义事件创建事件分派器，需要首先在类中注册事件名称，然后创建同名的方法。

例如下面这段代码所示：

```
class MyEventDispatcher(EventDispatcher):
    def __init__(self, **kwargs):
        self.register_event_type('on_test')
        super(MyEventDispatcher, self).__init__(**kwargs)

    def do_something(self, value):
        # when do_something is called, the 'on_test' event will be
        # dispatched with the value
        self.dispatch('on_test', value)

    def on_test(self, *args):
        print "I am dispatched", args
```

附加回调

要利用一个事件，必须要对其绑定回调。当事件被分派的时候，该特定事件相关的参数将被用于调用回调。

回调可以使Python中能进行调用的任意内容，函数或者方法都可以，但一定要确保回调要接收事件发出的参数。最安全的常规做法是接收* args参数，将所有参数都存放成一个参数列表。

例如：

```
def my_callback(value, *args):
    print "Hello, I got an event!", args

ev = MyEventDispatcher()
ev.bind(on_test=my_callback)
ev.do_something('test')
```

请阅读参考 kivy.event.EventDispatcher.bind() 方法的文档来查看更多附加调用相关的样例。

简介属性Properties

> 群友 `十月的天空` 提示：attribute 和 property 都翻译成了 属性， 字面上确实没错，但是读起来就莫名其妙了。 从本质上讲 property是 kivy 特色，个人理解 property 包含于 attitude 而且绑定 widget 类，是否可以翻译成 `控件属性` 或者 `构件属性` 之类。

`控件属性`（Properties）是定义和绑定事件的一种很赞的办法。关键就是属性能生成事件，这样当你的某个对象中有一个属性（attribute）发生改变的时候，所有引用该属性（attribute）的 `控件属性`（Properties）都会被自动更新

译者注：我汉语词汇量匮乏了，很痛苦，这里说不明白了，所以就放了原文的单词作为对比，避免混淆了。

针对你要处理的数据类型，存在很多种不同类型的`控件属性`性（Properties）：

- 字符串属性StringProperty
- 数值属性NumericProperty
- 有界数值属性BoundedNumericProperty
- 对象属性ObjectProperty
- 词典属性DictProperty
- 列表属性ListProperty
- 选项属性OptionProperty
- 别名属性AliasProperty
- 布尔属性BooleanProperty
- 引用属性ReferenceListProperty

属性声明

要声明`控件属性`（Properties），必须要在类的层次上进行声明。接下来这个类才能在你创建对象的时候对真是的属性（attributes）进行实例化。此`控件属性`（Properties）非彼属性（attributes），`控件属性`Properties是根据你的attributes来创建事件的机制，例如：

```
class MyWidget(Widget):
    text = StringProperty('')
```

当覆盖初始化方法**init**的时候，一定要接收**kwargs做参数，并且一定要用super()来调用基类的初始化方法**init**，传递自定义类的实例过去：

```
def __init__(self, **kwargs):
    super(MyWidget, self).__init__(**kwargs)
```

分派属性事件

Kivy的`控件属性`Property，默认提供了一个on_事件。在属性被改变的时候，就会调用这个事件了。

特别注意

如果属性的新值与当前已有的值相等，那么on_事件就不会被调用了。

例如下面这段代码：

```
class CustomBtn(Widget):

    pressed = ListProperty([0, 0])

    def on_touch_down(self, touch):
        if self.collide_point(*touch.pos):
            self.pressed = touch.pos
            return True
        return super(CustomBtn, self).on_touch_down(touch)

    def on_pressed(self, instance, pos):
        print ('pressed at {pos}'.format(pos=pos))
```

上面代码的第三行中：

```
pressed = ListProperty([0, 0])
```

这一句中，基于ListProperty定义了一个pressed按下的属性，默认值是[0,0]。从这往后，只要这个属性被改变了，on_presses事件就会被调用。

第五行有如下代码：

```
def on_touch_down(self, touch):
    if self.collide_point(*touch.pos):
        self.pressed = touch.pos
        return True
    return super(CustomBtn, self).on_touch_down(touch)
```

这部分代码覆盖了控件类的_touch_down()方法。这段代码中，用控件对touch触碰的位置进行了检测。

如果touch的位置在控件范围内，就把pressed的值改变成touch.pos这个值，然后返回True，这表明程序已经处理好了这个touch了，就把这个touch消耗掉了，不用再传播了。

如果touch的位置在控件外部，就通过super(...)了调用原始事件，并返回结果。这就和常规情况一样了，touch事件会被继续传递下去。

在第十一行：

```
def on_pressed(self, instance, pos):
    print ('pressed at {pos}'.format(pos=pos))
```

这里定义了一个on_pressed函数，只要属性值发生改变了，这个函数就会被调用。

特别注意

这个on_事件是在类内定义属性的位置被调用。在定义该属性的类之外，若要监控/观察一个属性的任何变动，就必须丢这个属性进行bind绑定操作。

只能读取到一个控件实例的时候，要怎么去监控属性的变化呢？这时候用**bind**绑定一下属性就行了：

```
your_widget_instance.bind(property_name=function_name)
```

例如下面这段代码：

```
class RootWidget(BoxLayout):

    def __init__(self, **kwargs):
        super(RootWidget, self).__init__(**kwargs)
        self.add_widget(Button(text='btn 1'))
        cb = CustomBtn()
        cb.bind(pressed=self.btn_pressed)
        self.add_widget(cb)
        self.add_widget(Button(text='btn 2'))

    def btn_pressed(self, instance, pos):
        print ('pos: printed from root widget: {pos}'.format(pos=.pos))
```

如果运行上面这段代码，会发现有两次print输出语句出现在控制台中。第一个是来自_pressed事件，在CustomBtn类内部调用；另外一次print是来自我们用bind绑定到了属性变化上的btn_pressed函数。

（译者注：在用bind绑定了之后，属性的变化都会通过bind的函数被看到了。）

两个函数都被调用的原因很简单。Bind绑定操作并不意味着覆盖。这两个函数同时保留就冗余了，所以一般情况你只选择一个来对属性变化进行监听/反应就行了。

你还得注意一下传递给on_事件或者绑定到属性的函数的参数。

```
def btn_pressed(self, instance, pos):
```

第一个参数是self，这就是该函数所在类本身的一个实例。也可以用内联函数，如下代码所示：

```
cb = CustomBtn()

def _local_func(instance, pos):
    print ('pos: printed from root widget: {pos}'.format(pos=pos))

cb.bind(pressed=_local_func)
self.add_widget(cb)
```

第一个参数是定义属性的类的实例。第二个参数是一个值，这个值是属性的新值。 上面那一段只是代码片段，下面这一段代码是完整的样例了，可以复制粘贴到编辑器里面然后测试一下：

```
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout
from kivy.properties import ListProperty

class RootWidget(BoxLayout):

    def __init__(self, **kwargs):
        super(RootWidget, self).__init__(**kwargs)
        self.add_widget(Button(text='btn 1'))
        cb = CustomBtn()
        cb.bind(pressed=self.btn_pressed)
        self.add_widget(cb)
        self.add_widget(Button(text='btn 2'))

    def btn_pressed(self, instance, pos):
        print ('pos: printed from root widget: {pos}'.format(pos=pos))

class CustomBtn(Widget):

    pressed = ListProperty([0, 0])

    def on_touch_down(self, touch):
        if self.collide_point(*touch.pos):
            self.pressed = touch.pos
            # we consumed the touch. return False here to propagate
            # the touch further to the children.
            return True
        return super(CustomBtn, self).on_touch_down(touch)

    def on_pressed(self, instance, pos):
        print ('pressed at {pos}'.format(pos=pos))

class TestApp(App):

    def build(self):
        return RootWidget()

if __name__ == '__main__':
    TestApp().run()
```

运行上面这段完整的样例代码，会得到如下图所示的输出：



咱们这个CustomBtn没有做任何视觉上的调整，所以就是个大黑块。你可以触摸/点击这个黑色的区域，来看看控制台里面的输出。

复合属性

定义一个别名属性AliasProperty的时候，通常要定义一个getter函数和一个setter函数，前者用来读取值，后者用来设定值。这时候，你就得通过bind绑定参数来确定好getter和setter函数的调用时间。

例如下面这段代码：

```
cursor_pos = AliasProperty(_get_cursor_pos, None, bind=(
    'cursor', 'padding', 'pos', 'size', 'focus',
    'scroll_x', 'scroll_y'))
'''Current position of the cursor, in (x, y).

:attr:`cursor_pos` is a :class:`~kivy.properties.AliasProperty`, read-only.
'''
```

这里的cursor_pos（光标位置的意思）就是一个别名属性AliasProperty，它有一个getter函数，名为_get_cursor_pos()，然后没有设置setter函数，这就说明这个属性是只读的。

最末尾那一段的bind参数的意思是，当在bind=这个等号后括号内的属性中有任意的一个发生变化，都会分派on_cursor_pos事件。

Title: Kivy Input management Date: 2017-02-12 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：输入管理

英文原文

译者前言

这一章节比上一章节翻译的还差，最近睡眠不太好，术后恢复比较差，大家凑合看看，看不下去给指出来一下比较不好理解和绕的地方，以及错误的地方，我一定即时修改。

输入体系

Kivy能处理绝大多数的输入类型：鼠标，触摸屏，加速器，陀螺仪等等。并且针对以下平台能够处理多点触控的原生协议：Tuio, WM_Touch, MacMultitouchSupport, MT Protocol A/B 以及 Android。（译者注：第一个TUIO应该是通用多点触控，第二个怀疑是WindowsMobile的，第三个是苹果的多点触控，第四个不知道是啥，最后一个是Android的。）

整体上输入体系的结构概括起来如下所示：

```
Input providers -> Motion event -> Post processing -> Dispatch to Window

输入源 -> 动作事件 -> 事后处理 -> 分派到窗口
```

所有输入事件的类是 `MotionEvent` 。这个类生成两种事件：

- Touch触控事件：包含位置信息，至少X和Y坐标位置的一种Motion动作事件。所有这种Touch事件都通过控件树进行分派。
- Non-Touch非触控事件：其余的各种事件。例如加速度传感器就是一个持续的事件，不具有坐标位置。这一事件没有起止，一直在发生。这类的事件都不通过控件树来分派。

Motion动作事件是由 `InputProvider` 生成的。 InputProvider这个类就是负责读取输入事件，这些输入事件的来源可以是操作系统，网络或者其他的应用程序。如下这几个都是已有的输入源：

- `TuioMotionEventProvider` ：创建一个UDP服务端，侦听TUIO/OSC信息。
- `WM_MotionEventProvider` ：使用Windows API来读取多点触控信息并发送给Kivy。
- `ProbeSysfsHardwareProbe` ：在Linux中，遍历连接到计算机的所有硬件，并为找到的每个多点触摸设备附加一个多点触摸输入提供程序。

- 还有很多很多啦！

当你写一个应用程序的时候，就不用再去重造一个输入源了。Kivy会自动检测可用的硬件。然而，如果你想要支持某些特殊定制的专门硬件，就可能得对Kivy的配置进行一下调整才行。

在新建的Motion动作事件被传递给用户之前，Kivy会先对输入进行处理。Kivy会对每一个动作事件进行分析来检查和纠正错误输入，也是保证能提供有意义的解释，比如：

- 根据姿势和持续时间来检测双击或三次点击；
- 在硬件设备精度不佳的情况下提高事件精确度；
- 原生触摸硬件若在近似相同位置发送事件则降低生成事件数量。

经过上面这些步骤之后，这个Motion动作事件就会被分派给对应的窗口。正如之前解释过的，并非所有事件都分派给整个控件树，程序窗口要对事件进行过滤筛选。对于一个给定的事件：

- 如果仅仅是一个Motion动作事件，那它就会被分派给 `on_motion()`；
- 如果是一个Touch事件，这个触摸控件的坐标位置（x,y)（范围在0-1）会被调整到与窗口尺寸（宽高）相适应，然后对应发给下面这些方法：
    - `on_touch_down()`
    - `on_touch_move()`
    - `on_touch_up()`

Motion动作事件的属性

你用的硬件和输入源可能允许你能获取到更多信息。比如一个Touch触摸输入不仅有坐标位置（x,y），还可能有压力强度信息，触摸范围大小，加速度矢量等等。

在Motion动作事件中，有一个字符串作为profile属性，用于说明该事件内都有那些可用的效果。假如咱们有下面这样的一个 `on_touch_move` 方法：

```
def on_touch_move(self, touch):
    print(touch.profile)
    return super(..., self).on_touch_move(touch)
```

在控制台的打印输出可能是：

```
['pos', 'angle']
```

特别注意

很多人可能会把这里Motion事件的Profile属性的名字与对应的Property属性弄混。一定要注意，可用Profile属性中存在 `angle` ，并不意味着Touch事件对象也必须有一个 `angle` 的Property属性。

对应profile属性 `'pos'` ，property属性中有位置信息 `pos` ， `x` , `y` 。profile属性 `angle` ，property属性对应的是有角度 `a` 。刚刚我们就说了，对touchTouch事件来说，profile属性中按照惯例是必须有位置属性 `pos` 的，但不一定有角度属性 `angle` 。对角度属性 `angle` 是否存在，可以用下面的方法来检测一下：

```
def on_touch_move(self, touch):
    print('The touch is at position', touch.pos)
    if 'angle' in touch.profile:
        print('The touch angle is', touch.a)
```

在 `motionevent` 文档中，可以找到所有可用profile属性的列表。

Touch事件

有一种特殊的 `MotionEvent` 动作事件 ，这种事件的 `is_touch` 方法返回的是True，这就是Touch事件。

所有的Touch事件，都默认就有X和Y的坐标信息，与窗口的宽度和高度相匹配。换句话说就是所有的Touch事件都有 `pos` 这一profile属性。

基本简介

默认情况下，Touch事件会被分派给所有当前显示的控件。也就是说无论这个Touch是否发生在控件的物理范围内，控件都会收到它。

如果你接触过其他的GUI框架，可能觉得这特点挺违背直觉的。一般的GUI框架里面，都是把屏幕分割成多个几何区域，然后只在发生区域内的控件才会被分派到触摸或者鼠标事件。

这个设定对触摸输入的情景来说就过于严格了。因为用手指划，之间点戳，还有长时间按，都可能会有偏移导致落到 用户希望进行交互的控件外的情景。

为了提供最大的灵活性，Kivy会把事件分派给所有控件，然后让控件来自行决定如何应对这些事件。如果你只希望在某个控件内对Touch事件作出反应，只需要按照如下方法进行一下检测：

```
def on_touch_down(self, touch):
    if self.collide_point(*touch.pos):
        # The touch has occurred inside the widgets area. Do stuff!
        pass
```

坐标位置

一旦你使用一个带有矩阵变换的控件，就一定要处理好Touch事件中的矩阵变换。例如 `Scatter` 这样的某些控件，自身会有矩阵变换，这就意味着Touch事件也必须用Scatter矩阵进行处理，这样才能正确地把Touch事件的位置分派给Scatter的子控件。

- 从上层空间到本地空间获取坐标： `to_local()`
- 从本地空间到上层空间获取坐标： `to_parent()`
- 从本地空间到窗口空间获取坐标： `to_window()`
- 从窗口空间到本地空间获取坐标： `to_widget()`

一定要使用上面方法当中的某一种来确保内容坐标系适配正确。然后下面这段代码里是Scatter的实现：

```
def on_touch_down(self, touch):
    # push the current coordinate, to be able to restore it later
    # 这里用push先把当前的坐标位置存留起来，以后就还可以恢复到这个坐标
    touch.push()

    # transform the touch coordinate to local space
    # 接下来就是把Touch的坐标转换成本地空间的坐标
    touch.apply_transform_2d(self.to_local)

    # dispatch the touch as usual to children
    # the coordinate in the touch is now in local space
    # 转换之后把这个Touch事件按照惯例分派给子控件
    # Touch事件的坐标位置现在就是本地空间的了
    ret = super(..., self).on_touch_down(touch)

    # whatever the result, don't forget to pop your transformation
    # after the call, so the coordinate will be back in parent space
    #无论结果如何，一定记得把这个转换用pop弹出
    # 之后，坐标就又恢复成上层空间的了
    touch.pop()

    # return the result (depending what you want.)
    # 最后就是返回结果了
    return ret
```

Touch事件的形状

If the touch has a shape, it will be reflected in the ‘shape’ property. Right now, only a `ShapeRect` can be exposed:

如果你的Touch事件有某个形状，这个信息会反映在 `shape` 这一property属性中。目前能用的就是一个 `ShapeRect` ：

```
from kivy.input.shape import ShapeRect
def on_touch_move(self, touch):
    if isinstance(touch.shape, ShapeRect):
        print('My touch have a rectangle shape of size',
            (touch.shape.width, touch.shape.height))
    # ...
```

双击

A double tap is the action of tapping twice within a time and a distance. It's calculated by the doubletap post-processing module. You can test if the current touch is one of a double tap or not:

双击是一种特定动作，在一小段时间和很短的一小段特定距离内敲击两下。双击的计算识别是通过一个双击后处理模块来实现的。可以用如下代码来检测当前的Touch是否是双击动作中的一下：

```
def on_touch_down(self, touch):
    if touch.is_double_tap:
        print('Touch is a double tap !')
        print(' - interval is', touch.double_tap_time)
        print(' - distance between previous is', touch.double_tap_distance)
    # ...
```

三次点击

A triple tap is the action of tapping thrice within a time and a distance. It's calculated by the tripletap post-processing module. You can test if the current touch is one of a triple tap or not:

三次点击和双击的概念类似，只不过是变成了点击三次。这个是通过一个三次点击后处理模块来计算识别的。可以用如下代码来检测当前的Touch是否是三次点击动作中的一下：

```
def on_touch_down(self, touch):
    if touch.is_triple_tap:
        print('Touch is a triple tap !')
        print(' - interval is', touch.triple_tap_time)
        print(' - distance between previous is', touch.triple_tap_distance)
    # ...
```

拖放事件

父控件可能会从`on_touch_down`中分派Touch事件到子控件，而不从`on_touch_move`或`on_touch_up`分派。这可能发生在某些特定情况知悉啊，比如一个Touch处于父控件的边界之外，这样父控件就会决定不对子控件通知这个Touch。

But you might want to do something in `on_touch_up`. Say you started something in the `on_touch_down` event, like playing a sound, and you'd like to finish things on the `on_touch_up` event. Grabbing is what you need.

不过有可能你还是得处理一下`on_touch_up`。比方说，你开始是`on_touch_down`事件，假设是按下播放语音之类的，然后你希望当手指抬起的时候`on_touch_up`事件发生的时候就结束任务。这时候就需要有Grab拖放事件了。

When you grab a touch, you will always receive the move and up event. But there are some limitations to grabbing:

拖放一个Touch的时候，总会收到移动和抬起事件。但对拖放有如下的限制：

- 至少会两次收到这个事件：一次是从父控件正常收到的事件，还有一次是从窗口获取的Grab拖放事件。
- 有可能你没有进行拖放，但还是会收到一个拖放Touch事件：这可能是因为在子控件处于拖放状态时，父控件发来了一个Touch事件。
- 在拖放状态下，Touch事件的坐标不会转换成控件空间的坐标，因为这个Touch事件是直接来自窗口的。所以要手动将坐标转换到本地空间。

下面这段代码展示了对拖放的使用：

```
def on_touch_down(self, touch):
    if self.collide_point(*touch.pos):

        # if the touch collides with our widget, let's grab it
        touch.grab(self)

        # and accept the touch.
        return True

def on_touch_up(self, touch):
    # here, you don't check if the touch collides or things like that.
    # you just need to check if it's a grabbed touch event
    if touch.grab_current is self:

        # ok, the current touch is dispatched for us.
        # do something interesting here
        print('Hello world!')

        # don't forget to ungrab ourself, or you might have side effects
        touch.ungrab(self)

        # and accept the last up
        return True
```

Touch事件管理

想要了解更多Touch事件如何控制以及如何在控件之间传递，可以阅读一下Widget touch event bubbling这部分内容。

Title: Kivy Widgets Date: 2017-02-26 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：控件

英文原文

控件简介

控件 `Widget` 是 Kivy 图形界面中的基本元素。控件提供了一个画布 `Canvas` ，这是用来在屏幕上进行绘制的。控件接收事件，并且对事件作出反应。想要对 控件 `Widget` 进行更深入的了解，可以去看看这个模块的文档。

操作控件树

Kivy 以树的形式来组织控件。你的应用程序会有一个根控件，通常会含有若干的子控件 `children` ，这些子控件还可以有自己的子控件。一个控件的子控件会以 `children` 属性的形式表述，这个属性是 Kivy 中的一个列表属性 `ListProperty`

可以用一下方法来操作控件树：

- `add_widget()`：添加一个控件作为子控件；
- `remove_widget()`：从子控件列表中去掉一个控件；
- `clear_widgets()`：清空一个控件的所有子控件。

例如下面的代码，就是在一个盒式布局 BoxLayout 中添加一个按钮：

```
layout = BoxLayout(padding=10)
button = Button(text='My first button')
layout.add_widget(button)
```

这个按钮就添加到布局当中去了：按钮的 parent 属性会被设置为这个布局；这个按钮也会被添加到布局中的子控件列表。要把这个按钮从这个布局中删掉也很简单：

```
layout.remove_widget(button)
```

移除了之后，这个按钮的 parent 属性就会被设置为 None，也会被从布局的子控件列表中移除。

要是想清空一个控件中的所有自科技，那就用 `clear_widgets()` 方法就可以了：

```
layout.clear_widgets()
```

特别注意

千万别自己手动去操作子控件列表，除非你确定自己掌控得非常深入透彻。因为控件树是和绘图树联系在一起的。例如，如果你添加了一个控件到子控件列表，但没有添加这个新子控件的画布到绘图树上，那么就会出现这种情况：这个控件确实成了一个子控件，但是屏幕上不会显示出来。此外，如果你后续使用添加、移除、清空控件这些操作，可能还会遇到问题。

遍历控件树

控件类实例的子控件 `children` 列表属性中包含了所有的子控件。所以可以用如下的方式来进行遍历：

```
root = BoxLayout()
# ... add widgets to root ...
for child in root.children:
    print(child)
```

然而，这样的操作可得谨慎使用。如果你要用之前一节中提到的方法来修改这个子控件列表电话，请一定用下面这种方法来做一下备份：

```
for child in root.children[:]:
    # manipulate the tree. For example here, remove all widgets that have a
    # width
    if child.width  100:
        root.remove_widget(child)
```

默认情况下，控件是不会对子控件的尺寸/位置进行改变的。位置属性 `pos` 是屏幕坐标系上的绝对值（除非你使用相对布局 `relativelayout` ，这个以后再说），而尺寸属性 `size` 就是一个绝对的尺寸大小。

控件索引Z

控件绘制的顺序，是基于各个控件在控件树中的位置。添加控件方法 `add_widget` 可以接收一个索引参数，这样就能指定该新增控件在控件树中的位置。

```
root.add_widget(widget, index)
```

索引值小的控件会被绘制在索引值大的控件之上。一定要记住，默认值是 0 ，所以后添加的控件总会在所有控件的最顶层，除非指定了索引值。

整理布局

布局 `layout` 是一种特别的控件，它会控制自己子控件的尺寸和位置。有各种不同的布局，这些布局分别为子控件提供拜托你个的自动组织整理。这些布局使用尺寸推测 `size_hint` 和位置推测 `pos_hint` 这两个属性来决定子控件 `children` 的尺寸 `size` 和 位置 `pos` 。

盒式布局 **BoxLayout**: 所有控件充满整个空间，以互相挨着的方块的方式来分布，横着或者竖着排列都可以。子控件的 size_hint 属性可以用来改变每个子控件的比例，也可以设置为固定尺寸。

Box Layout

horizontal

Grid Layout

rows = 3

Stack Layout

lr-tb

Anchor Layout

anchor_x = left
anchor_y = top



网格布局 **GridLayout**: 以一张网格的方式来安排控件。你必须指定好网格的维度，确定好分成多少格，这样 Kivy 才能计算出每个元素的尺寸并且确定如何安排这些元素的位置。

栈状布局 **StackLayout**: 挨着放一个个控件，彼此邻近，在某一个维度上有固定大小，而使它们填充整个空间。 这适合用来显示相同预定义大小的子控件。

锚式布局 **AnchorLayout**: 一种非常简单的布局，只关注子控件的位置。 将子控件放在布局的边界位置。 不支持size_hint。

浮动布局 **FloatLayout**: 允许放置具任意位置和尺寸的子控件，可以是绝对尺寸，也可以是相对布局的相对尺寸。 默认的 size_hint（1，1）会让每个子控件都与整个布局一样大，所以如果你多个子控件就要修改这个值。可以把 size_hint 设置成 (None, None)，这样就可以使用 size 这个绝对尺寸属性。控件也支持 pos_hint，这个属性是一个 dict 词典，用来设置相对布局的位置。

相对布局 **RelativeLayout**: 和浮动布局 FloatLayout 差不多，不同之处在于子控件的位置是相对于布局空间的，而不是相对于屏幕。

想要深入理解各种布局的话，可以仔细阅读各种文档。

`size_hint` 和 `pos_hint`：

- `floatlayout`
- `boxlayout`
- `gridlayout`
- `stacklayout`
- `relativelayout`
- `anchorlayout`

`size_hint` 是一个 引用列表属性 `ReferenceListProperty` ，包括 `size_hint_x` 和 `size_hint_y` 两个变量。接收的变量值是从0到1的各种数值，或者 None， 默认值为 (1, 1)。这表示如果控件处在布局之内，布局将会在两个方向分配全部尺寸（相对于布局大小）给该控件。

举个例子，设置 `size_hint` 为 (0.5, 0.8)，就会给该控件 `Widget` 分配布局 `layout` 内50% 宽，80% 高的尺寸。

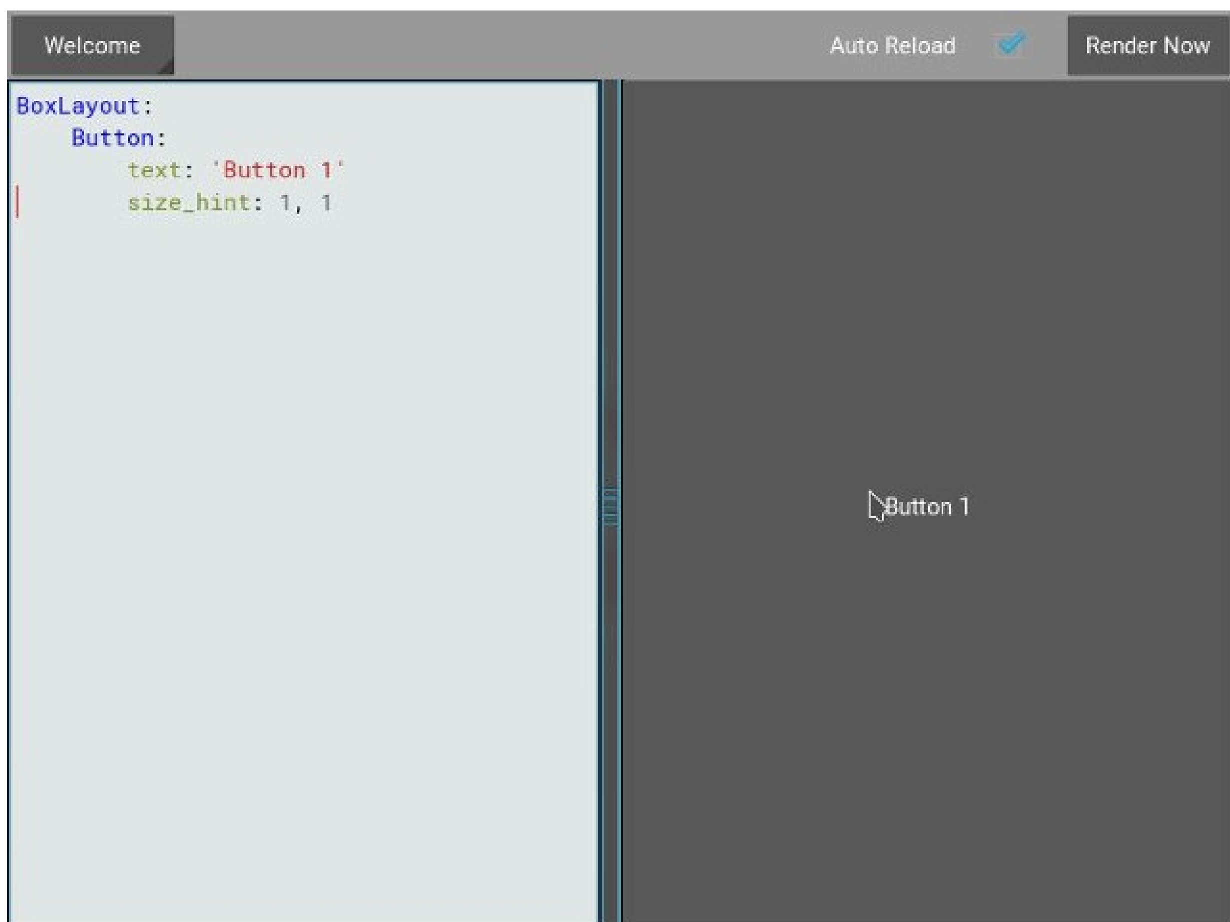
例如下面这个例子：

```
BoxLayout:
    Button:
        text: 'Button 1'
        # default size_hint is 1, 1, we don't need to specify it explicitly
        # however it's provided here to make things clear
        size_hint: 1, 1
```

加载 Kivy 目录：

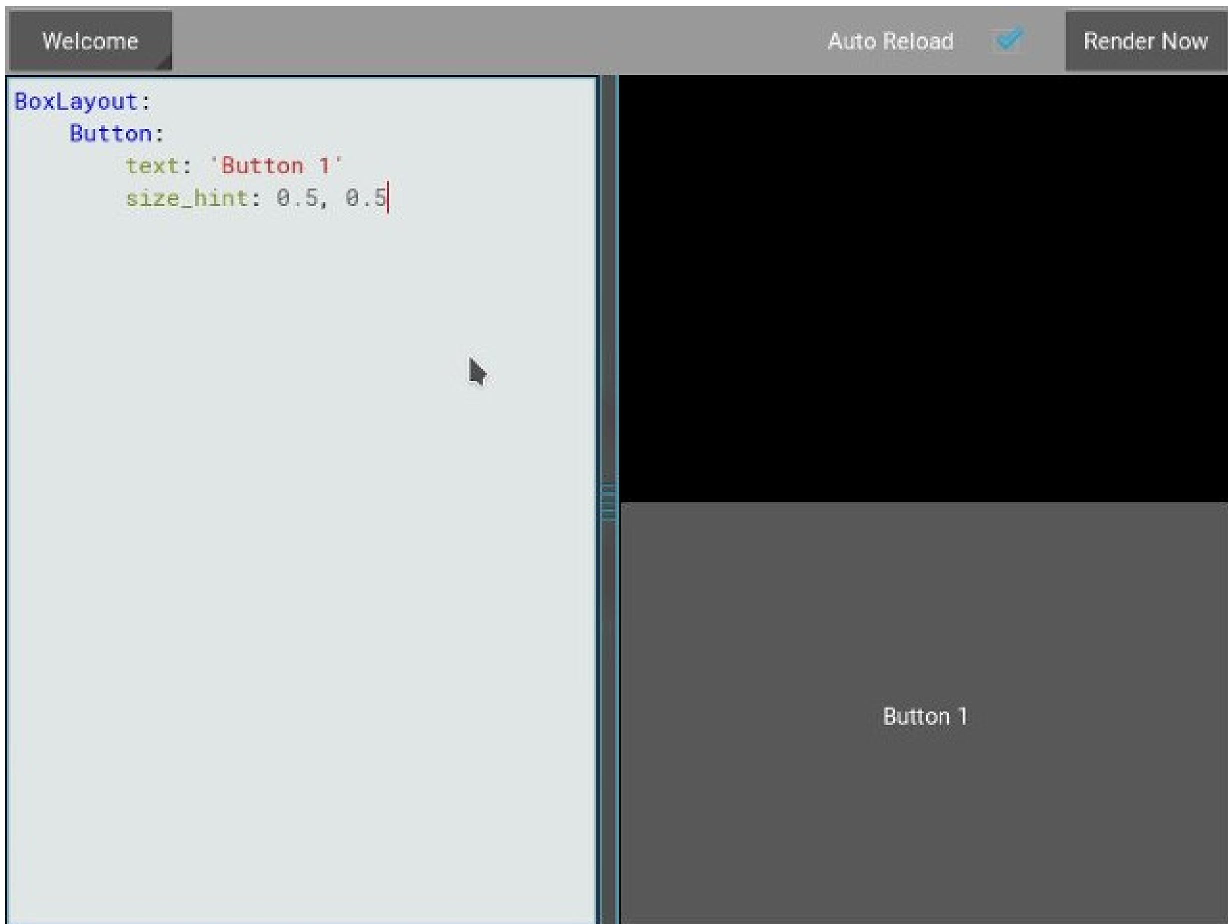
```
cd $KIVYDIR/examples/demo/kivycatalog
python main.py
```

把上面代码中的 $KIVYDIR 替换成你的 Kivy 安装位置。在左边点击标注有 Box Layout 的按钮。 然后将上面的代码粘贴到窗口右侧的编辑器内。



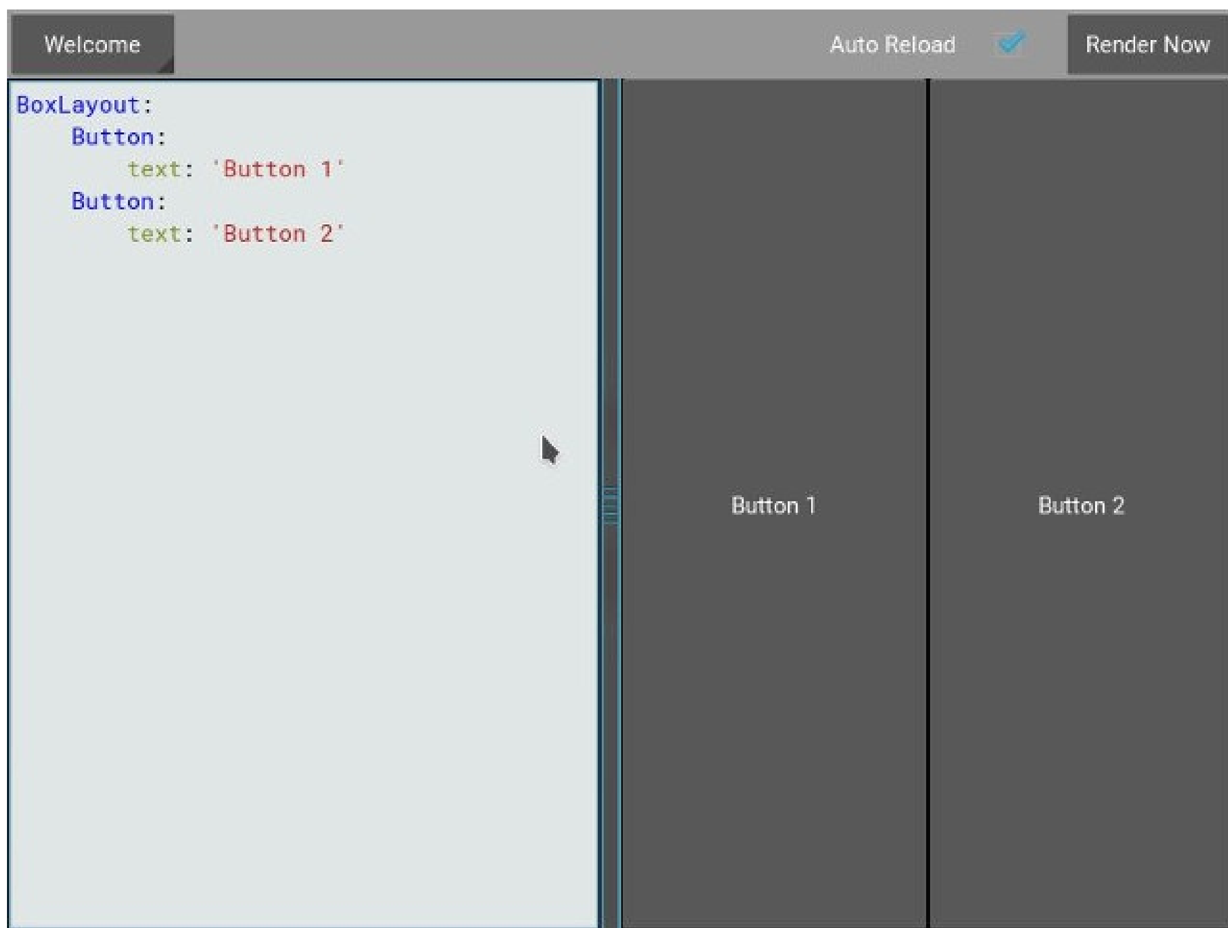
然后你就可以看到上图这样的界面了，这个按钮 Button 会占据整个布局尺寸 `size` 的 100%。

修改 size_hint_x / size_hint_y 为 .5 这就会把控件 Widget 调整为布局 layout 的50% 宽度 width /高度 height 。

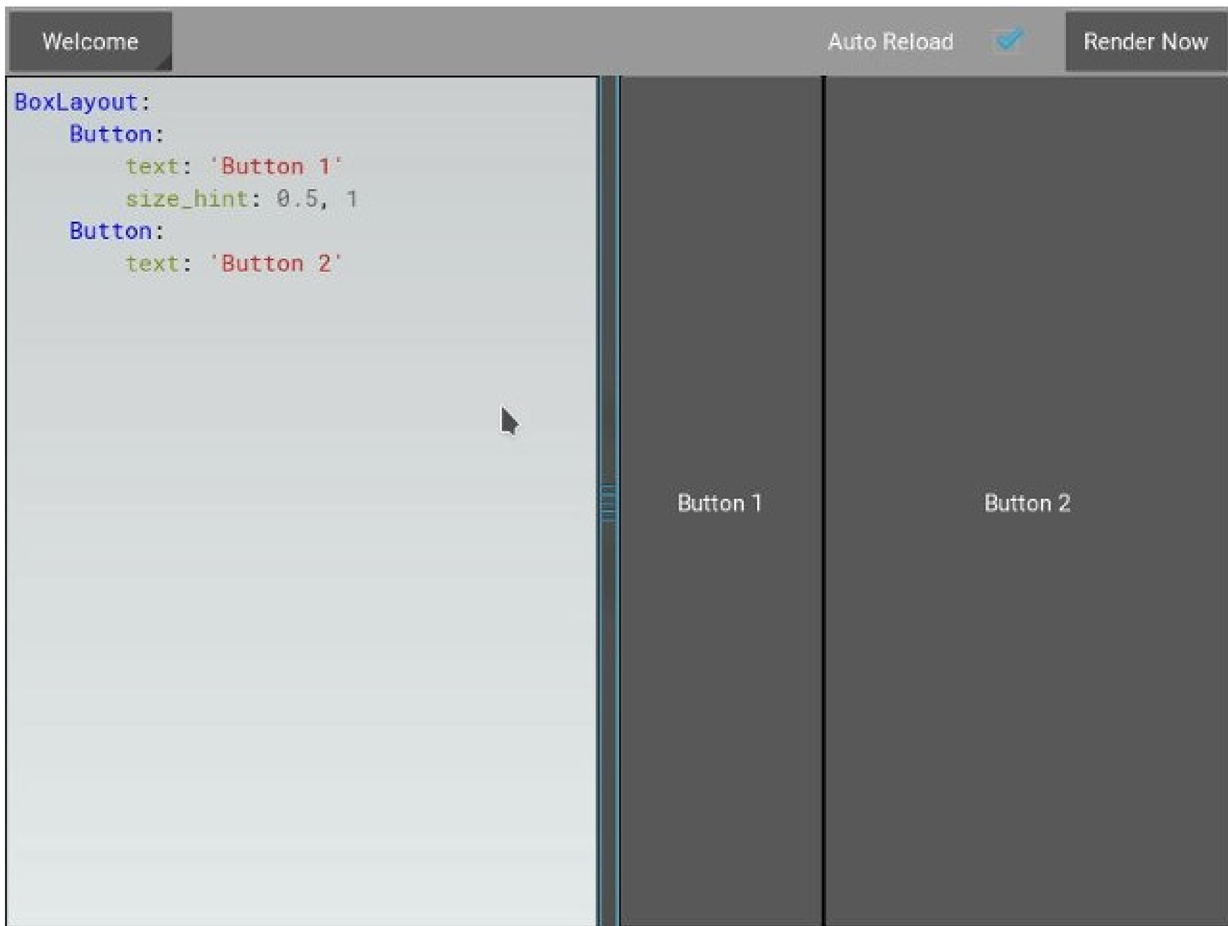


这时候效果如上图所示，虽然我们已经同时指定了 size_hint_x 和 size_hint_y 为 .5，但似乎只有对 size_hint_x 的修改起作用了。这是因为在盒式布局 boxlayout 中，当 orientation 被设置为竖直方向（vertical） 的时候， size_hint_y 由布局来控制，而如果 orientation 被设置为水平方向（horizontal）的时候， size_hint_x 由布局来控制，所以这些情况手动设定就无效了。 这些受控维度的尺寸，是根据子控件 children 在 盒式布局 boxlayout 中的总编号来计算的。在上面的例子中，这个子控件的 size_hint_y 是受控的(.5/.5 = 1)。所以，这里控件就占据了上层布局的整个高度。

接下来咱们再添加一个按钮 Button 到这个 布局 layout 看看有什么效果。



盒式布局 `boxlayout` 默认对其所有的子控件 `children` 分配了等大的空间。在咱们这个例子里面，比例是50-50，因为有两个子控件 `children` 。那么接下来咱们就对其中的一个子控件设置一下 size_hint，然后看看效果怎么样。



从上图可以看出，如果一个子控件有了一个指定的 `size_hint` ，这就会决定该控件 `Widget` 使用盒式布局 `boxlayout` 提供的空间中的多大比例，来作为自己的尺寸 `size` 。在我们这个例子中，第一个按钮 `Button` 的 `size_hint_x` 设置为了 .5。那么这个控件分配到的空间计算方法如下：

```
first child's size_hint divided by
first child's size_hint + second child's size_hint + ...n(no of children)

.5/(.5+1) = .333...
```

盒式布局 BoxLayout 的剩余宽度 `width` 会分配给另外的一个子控件 `children` 。在我们这个例子中，这就意味着第二个按钮 `Button` 会占据整个布局 `layout` 的 66.66% 宽度 `width` 。

修改 `size_hint` 探索一下来多适应一下吧。

如果你想要控制一个控件 `Widget` 的绝对尺寸 `size` ，可以把 `size_hint_x` / `size_hint_y` 当中的一个或者两个都设置成 None，这样的话该控件的宽度 `width` 和高度 `height` 的属性值就会生效了。

`pos_hint` 是一个词典 dict，默认值是空。相比于 `size_hint` ，布局对 `pos_hint` 的处理方式有些不同，不过大体上你还是可以对 `pos` 的各种属性设定某个值来设定控件 `Widget` 在父控件 `parent` 中的相对位置（可以设定的属性包括： `x` , `y` , `right` , `top` , `center_x` ,

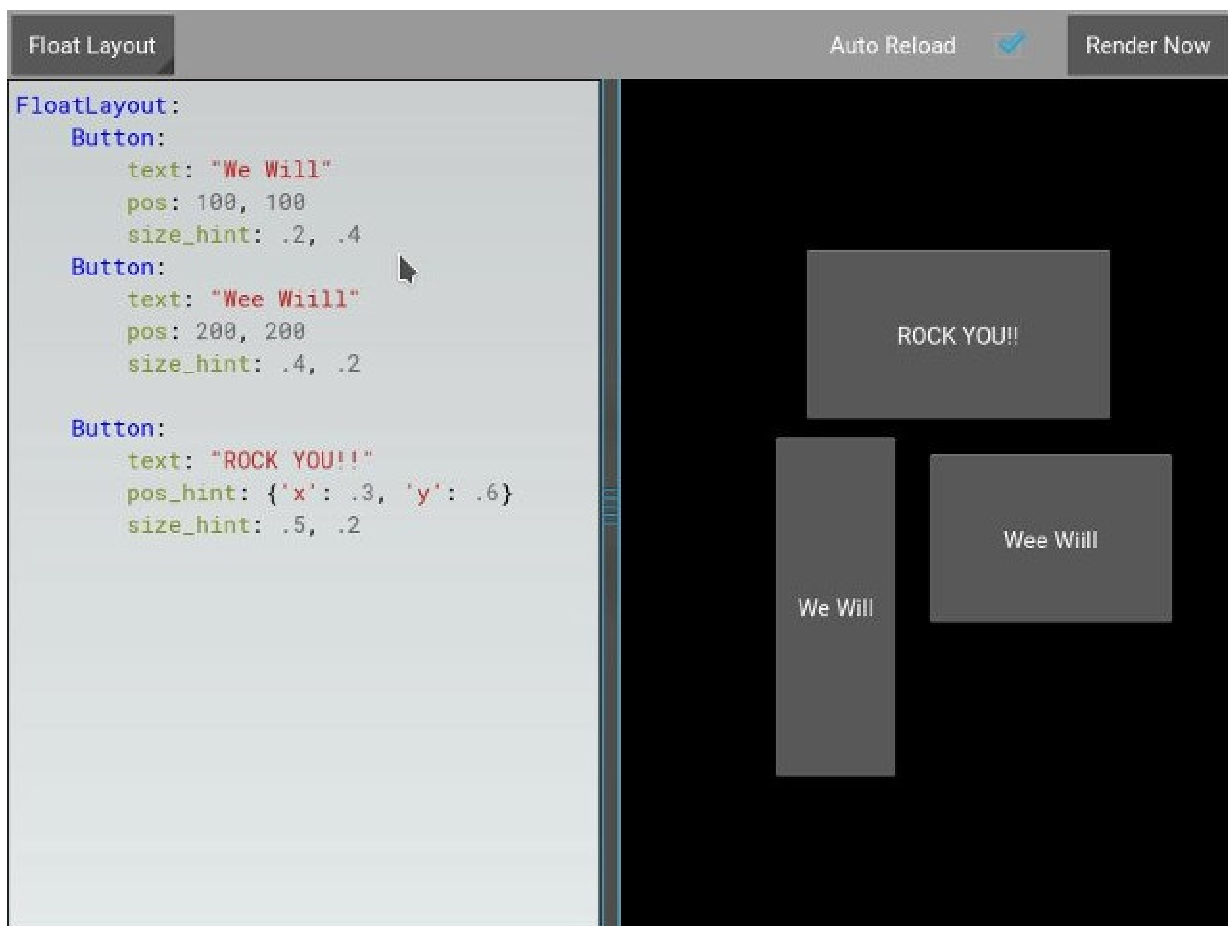
`center_y` ）。

咱们用下面 kivycatalog 中的代码来可视化地理解一下 `pos_hint` ：

```
FloatLayout:
    Button:
        text: "We Will"
        pos: 100, 100
        size_hint: .2, .4
    Button:
        text: "Wee Wiill"
        pos: 200, 200
        size_hint: .4, .2

    Button:
        text: "ROCK YOU!!"
        pos_hint: {'x': .3, 'y': .6}
        size_hint: .5, .2
```

这份代码的输出效果如下图所示：



说了半天 `size_hint` ，你不妨自己试试探索一下 `pos_hint` ，来理解一下这个属性对控件位置的效果。

给布局添加背景

关于布局，有一个问题经常被问道：

“怎么给一个布局添加背景图片/颜色/视频/等等......”

本来默认的各种布局都是没有视觉呈现的：因为布局不像控件，布局是默认不含有绘图指令的。不过呢，还是你可以给一个布局实例添上绘图指令，也就可以添加一个彩色背景了：

在 Python 中的实现方法：

```
from kivy.graphics import Color, Rectangle

with layout_instance.canvas.before:
    Color(0, 1, 0, 1) # green; colors range from 0-1 instead of 0-255
    self.rect = Rectangle(size=layout_instance.size,
                           pos=layout_instance.pos)
```

然而很不幸，这样只能在布局的初始化位置以布局的初始尺寸绘制一个矩形。所以还要对布局的尺寸和位置变化进行监听，然后对矩形的尺寸位置进行更新，这样才能保证这个矩形一直绘制在布局的内部。可以用如下方式实现：

```
with layout_instance.canvas.before:
    Color(0, 1, 0, 1) # green; colors range from 0-1 instead of 0-255
    self.rect = Rectangle(size=layout_instance.size,
                           pos=layout_instance.pos)

def update_rect(instance, value):
    instance.rect.pos = instance.pos
    instance.rect.size = instance.size

# listen to size and position changes
layout_instance.bind(pos=update_rect, size=update_rect)
```

在 kv 文件中：

```
FloatLayout:
    canvas.before:
        Color:
            rgba: 0, 1, 0, 1
        Rectangle:
            # self here refers to the widget i.e BoxLayout
            pos: self.pos
            size: self.size
```

上面的 Kv 文件中的生命，就建立了一个隐含的绑定：上面 Kv 代码中的最后两行保证了矩形的位置 `pos` 和尺寸 `size` 会在浮动布局 `floatlayout` 的位置 `pos` 发生变化的时候进行更新。

接下来咱们把上面的代码片段放进 Kivy 应用里面。

纯 Python 方法：

```
from kivy.app import App
from kivy.graphics import Color, Rectangle
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.button import Button

class RootWidget(FloatLayout):

    def __init__(self, **kwargs):
        # make sure we aren't overriding any important functionality
        super(RootWidget, self).__init__(**kwargs)

        # let's add a Widget to this layout
        self.add_widget(
            Button(
                text="Hello World",
                size_hint=(.5, .5),
                pos_hint={'center_x': .5, 'center_y': .5}))

class MainApp(App):

    def build(self):
        self.root = root = RootWidget()
        root.bind(size=self._update_rect, pos=self._update_rect)

        with root.canvas.before:
            Color(0, 1, 0, 1)  # green; colors range from 0-1 not 0-255
            self.rect = Rectangle(size=root.size, pos=root.pos)
        return root

    def _update_rect(self, instance, value):
        self.rect.pos = instance.pos
        self.rect.size = instance.size

if __name__ == '__main__':
    MainApp().run()
```

使用 Kv 语言：

```
from kivy.app import App
from kivy.lang import Builder

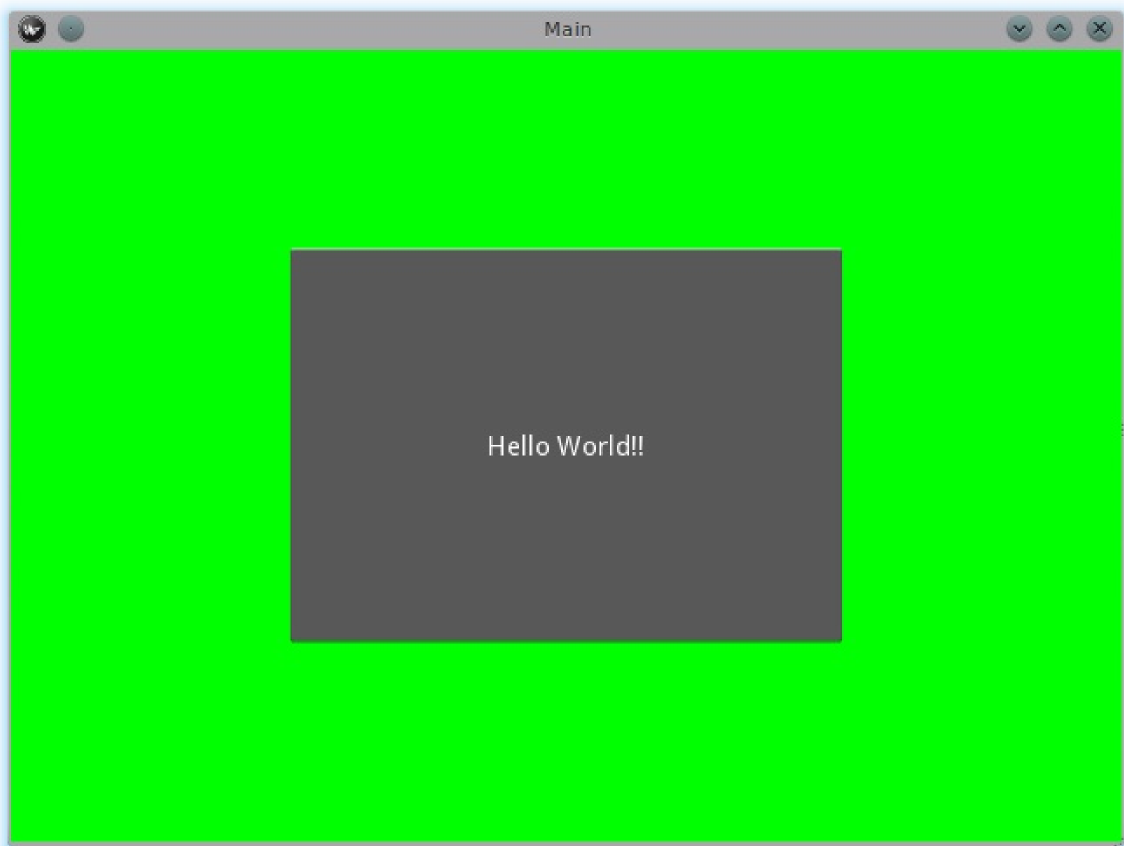

root = Builder.load_string('''
FloatLayout:
    canvas.before:
        Color:
            rgba: 0, 1, 0, 1
        Rectangle:
            # self here refers to the widget i.e FloatLayout
            pos: self.pos
            size: self.size
    Button:
        text: 'Hello World!!'
        size_hint: .5, .5
        pos_hint: {'center_x':.5, 'center_y': .5}
''')

class MainApp(App):

    def build(self):
        return root

if __name__ == '__main__':
    MainApp().run()
```

上面这两个应用的效果都如下图所示：



给自定义布局规则/类增加背景色

上面那一段中咱们对布局实例增加背景的方法，如果用到很多歌布局里面，那就很快变得特别麻烦了。要解决这种需求，就可以基于布局类 Layout 创建一个自定义的布局子类，给自定义的这个类增加一个背景。

使用 Python：

```
from kivy.app import App
from kivy.graphics import Color, Rectangle
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.image import AsyncImage

class RootWidget(BoxLayout):
    pass

class CustomLayout(FloatLayout):

    def __init__(self, **kwargs):
        # make sure we aren't overriding any important functionality
        super(CustomLayout, self).__init__(**kwargs)

        with self.canvas.before:
            Color(0, 1, 0, 1)  # green; colors range from 0-1 instead of 0-255
            self.rect = Rectangle(size=self.size, pos=self.pos)

        self.bind(size=self._update_rect, pos=self._update_rect)

    def _update_rect(self, instance, value):
        self.rect.pos = instance.pos
        self.rect.size = instance.size

class MainApp(App):

    def build(self):
        root = RootWidget()
        c = CustomLayout()
        root.add_widget(c)
        c.add_widget(
            AsyncImage(
                source="http://www.everythingzoomer.com/wp-content/uploads/2013/01/Mon
day-joke-289x277.jpg",
                size_hint= (1, .5),
                pos_hint={'center_x':.5, 'center_y':.5}))
        root.add_widget(AsyncImage(source='http://www.stuffistumbledupon.com/wp-conten
t/uploads/2012/05/Have-you-seen-this-dog-because-its-awesome-meme-puppy-doggy.jpg'))
        c = CustomLayout()
        c.add_widget(
            AsyncImage(
                source="http://www.stuffistumbledupon.com/wp-content/uploads/2012/04/G
et-a-Girlfriend-Meme-empty-wallet.jpg",
                size_hint= (1, .5),
                pos_hint={'center_x':.5, 'center_y':.5}))
        root.add_widget(c)
        return root

if __name__ == '__main__':
    MainApp().run()
```

使用 Kv 语言：

```
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.boxlayout import BoxLayout
from kivy.lang import Builder


Builder.load_string('''
<CustomLayout>
    canvas.before:
        Color:
            rgba: 0, 1, 0, 1
        Rectangle:
            pos: self.pos
            size: self.size

<RootWidget>
    CustomLayout:
        AsyncImage:
            source: 'http://www.everythingzoomer.com/wp-content/uploads/2013/01/Monday
-joke-289x277.jpg'
            size_hint: 1, .5
            pos_hint: {'center_x':.5, 'center_y': .5}
    AsyncImage:
        source: 'http://www.stuffistumbledupon.com/wp-content/uploads/2012/05/Have-you
-seen-this-dog-because-its-awesome-meme-puppy-doggy.jpg'
    CustomLayout
        AsyncImage:
            source: 'http://www.stuffistumbledupon.com/wp-content/uploads/2012/04/Get-
a-Girlfriend-Meme-empty-wallet.jpg'
            size_hint: 1, .5
            pos_hint: {'center_x':.5, 'center_y': .5}
''')

class RootWidget(BoxLayout):
    pass

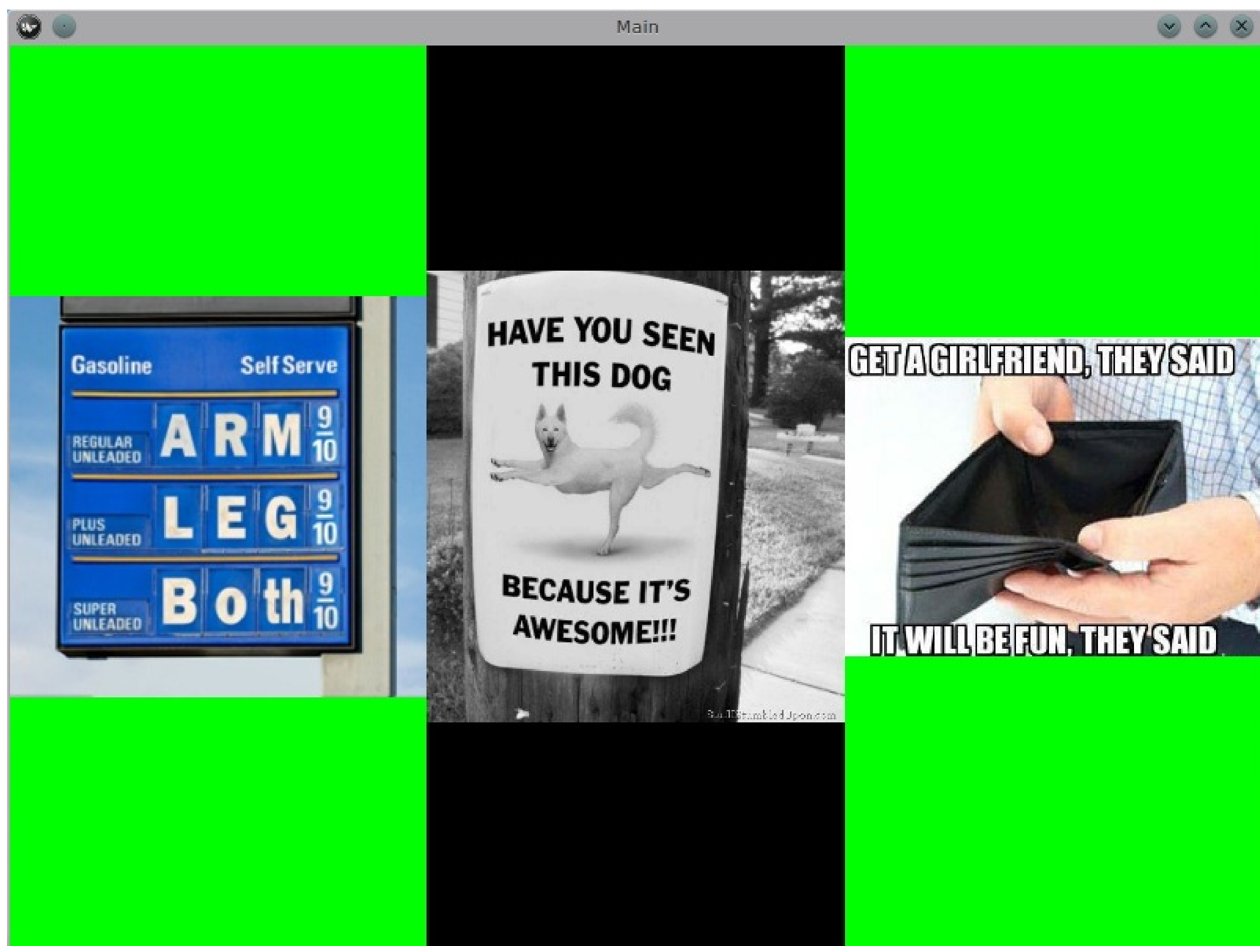
class CustomLayout(FloatLayout):
    pass

class MainApp(App):

    def build(self):
        return RootWidget()

if __name__ == '__main__':
    MainApp().run()
```

上面这两个应用的效果都如下图所示：



在自定义布局类中定义了背景之后，就是要确保在自定义布局的各个实例中使用到这个新特性。

首先，要在全局上增加一个图形或者颜色给内置的 Kivy 布局的背景，这就需要将所用布局的默认 Kv 规则进行覆盖。

就拿网格布局 GridLayout 举例吧：

```
<GridLayout>
    canvas.before:
        Color:
            rgba: 0, 1, 0, 1
        BorderImage:
            source: '../examples/widgets/sequenced_images/data/images/button_white.png'

            pos: self.pos
            size: self.size
```

接下来把这段代码放到一个 Kivy 应用里面：

```
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
from kivy.lang import Builder


Builder.load_string('''
<GridLayout>
    canvas.before:
        BorderImage:
            # BorderImage behaves like the CSS BorderImage
            border: 10, 10, 10, 10
            source: '../examples/widgets/sequenced_images/data/images/button_white.png
'
            pos: self.pos
            size: self.size

<RootWidget>
    GridLayout:
        size_hint: .9, .9
        pos_hint: {'center_x': .5, 'center_y': .5}
        rows:1
        Label:
            text: "I don't suffer from insanity, I enjoy every minute of it"
            text_size: self.width-20, self.height-20
            valign: 'top'
        Label:
            text: "When I was born I was so surprised; I didn't speak for a year and a
 half."
            text_size: self.width-20, self.height-20
            valign: 'middle'
            halign: 'center'
        Label:
            text: "A consultant is someone who takes a subject you understand and make
s it sound confusing"
            text_size: self.width-20, self.height-20
            valign: 'bottom'
            halign: 'justify'
''')

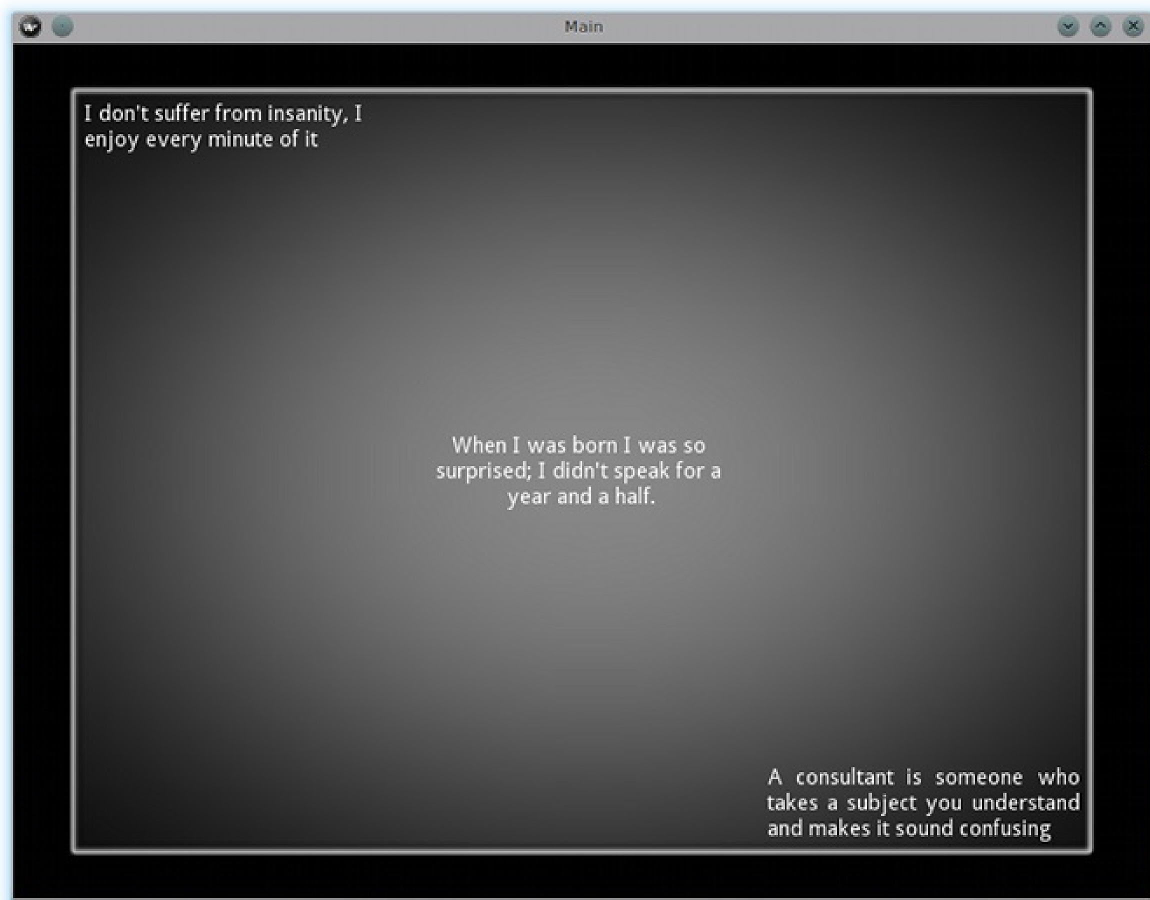
class RootWidget(FloatLayout):
    pass


class MainApp(App):

    def build(self):
        return RootWidget()

if __name__ == '__main__':
    MainApp().run()
```

效果大概如下图所示：



我们已经对网格布局 GridLayout 类的规则进行了覆盖，所以接下来在应用中使用这个类就都会显示那幅图片了。

动画背景怎么弄呢？

就像在矩形Rectangle/ 边界图像BorderImage /椭圆Ellipse/等里面添加设置绘图指令一样，可以用一个特定的纹理属性 texture ：

```
Rectangle:
    texture: reference to a texture
```

可以用下面的代码实现一个动画背景：

```
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.image import Image
from kivy.properties import ObjectProperty
from kivy.lang import Builder
```

```
Builder.load_string('''
<CustomLayout>
    canvas.before:
        BorderImage:
            # BorderImage behaves like the CSS BorderImage
            border: 10, 10, 10, 10
            texture: self.background_image.texture
            pos: self.pos
            size: self.size

<RootWidget>
    CustomLayout:
        size_hint: .9, .9
        pos_hint: {'center_x': .5, 'center_y': .5}
        rows:1
        Label:
            text: "I don't suffer from insanity, I enjoy every minute of it"
            text_size: self.width-20, self.height-20
            valign: 'top'
        Label:
            text: "When I was born I was so surprised; I didn't speak for a year and a
 half."
            text_size: self.width-20, self.height-20
            valign: 'middle'
            halign: 'center'
        Label:
            text: "A consultant is someone who takes a subject you understand and make
s it sound confusing"
            text_size: self.width-20, self.height-20
            valign: 'bottom'
            halign: 'justify'
''')


class CustomLayout(GridLayout):

    background_image = ObjectProperty(
        Image(
            source='../examples/widgets/sequenced_images/data/images/button_white_anim
ated.zip',
            anim_delay=.1))


class RootWidget(FloatLayout):
    pass


class MainApp(App):

    def build(self):
        return RootWidget()
```

```
if __name__ == '__main__':
    MainApp().run()
```

要理解这里到底发生了什么，得从第 13 行开始看：

```
texture: self.background_image.texture
```

这里是指定让边界图像 BorderImage 的纹理属性在背景图像 background_image 的纹理属性发生更新的时候进行同步更新。背景图像 background_image 属性的定义是在第 40 行：

```
background_image = ObjectProperty(...
```

这一句代码是将背景图像 background_image 设置成一个对象属性 `ObjectProperty` ，这样就可以在其中添加一个图形控件 `Image` 。图像控件有一个纹理属性（texture property）；在前面的 self.background_image.texture 这句代码中，就是建立了一个名为 texture 的到这个属性的引用。图形控件 `Image` 支持动画（animation）：随着动画的改变，图像的纹理会同步更新，在这个过程中，边界图像 BorderImage 指令的 texture 纹理属性也会同步更新。

（译者注：texture of BorderImage instruction，这里我对 instruction 的翻译应该是不太对的，不过我还没理清楚该怎么表述。）

也可以直接传递自定义数据到纹理属性 texture。更多细节可以参考纹理 `Texture` 的文档。

网状布局

嗯，看看这个过程如何扩展是很有趣的。

尺寸和位置度量

Kivy 的默认长度单位是像素 pixel，所有的尺寸和位置都用这个单位来表达。你也可以用其他单位来衡量，在跨平台多种设备的时候，这有助于实现更好的连续性体验（这些设备会把尺寸自动转换到像素）。

可用单位包括 `pt` ， `mm` ， `cm` , `inch` ， `dp` and `sp` 。可以在度量文档 `metrics` 中了解更多相关内容。

你还可以探索一下屏幕模块 `screen` 的用法，这个可以模拟出不同设备的屏幕，供测试应用。

使用屏幕管理器进行屏幕分割

如果你的应用程序要包含多个程序，那可能就需要从一个屏幕 `Screen` 到另一个屏幕 `Screen` 提供一个导航的通道。幸运的是，正好有一个屏幕管理器类 `ScreenManager` ，这个类允许你来定义分开的各个屏幕，设置屏幕管理器的 `TransitionBase` 就可以实现从一个屏幕到另一个屏幕的跳转导航。

Title: Kivy Graphics Date: 2017-02-12 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：图形

英文原文

译者前言

这一章节比前两章节简单很多，翻译的也比较顺了。

简介Canvas

Kivy中控件图形呈现是使用Canvas完成的，可以将其看作一个无限的绘图板，也是一组绘图指令。有很多种绘图指令都可以应用或者添加到你的Canvas伤，不过总体上分为两类：

- `context instructions` 环境指令
- `vertex instructions` 顶点指令

`context instructions` 环境指令不绘制任何图形，但会改变 `vertex instructions` 顶点指令的绘制结果。

Canvas都包含两个指令分支。分别是 `canvas.before` 和 `canvas.after` 这两种指令群。这两组指令分别在 `Canvas` 图形绘制前后执行。 绘制前的会被绘制的图形覆盖掉，绘制后的会覆盖在图形上层。这些指令都在用户对它们读取之后才会被创建。

要对一个控件添加Canvas绘图指令，需要使用Canvas环境指令：

```
class MyWidget(Widget):
    def __init__(self, **kwargs):
        super(MyWidget, self).__init__(**kwargs)
        with self.canvas:
            # add your instruction for main canvas here
            # 这里是增加一个座位主绘图的指令

        with self.canvas.before:
            # you can use this to add instructions rendered before
            # 这里可以在绘图之前添加指令

        with self.canvas.after:
            # you can use this to add instructions rendered after
            # 这里可以在绘图之后添加指令
```

环境指令

环境指令是用于操作opengl环境。 可以旋转，翻译和缩放画布。还可以附加纹理或更改绘图颜色。下面这段代码里面的是最常用到的更改颜色的指令，其他的环境指令也都很有用处：

```
with self.canvas.before:
    Color(1, 0, .4, mode='rgb')
```

绘图指令

绘图指令可简可繁，最简单的比如画一个多边形，更复杂的比如绘制网格或者贝塞尔曲线都可以：

```
with self.canvas:
   # draw a line using the default color
   # 用默认颜色画一条线
   Line(points=(x1, y1, x2, y2, x3, y3))

   # lets draw a semi-transparent red square
   # 接下来画一个半透明的红方块
   Color(1, 0, 0, .5, mode='rgba')
   Rectangle(pos=self.pos, size=self.size)
```

操作指令

有时候可能需要把之前添加到Canvas绘图上的指令进行更改或者删除，这可以有很多种办法，要根据具体需求来选择：

可以给指令创建一个引用然后对其进行更新：

```
class MyWidget(Widget):
    def __init__(self, **kwargs):
        super(MyWidget, self).__init__(**kwargs)
        with self.canvas:
            self.rect = Rectangle(pos=self.pos, size=self.size)

        self.bind(pos=self.update_rect)
        self.bind(size=self.update_rect)

    def update_rect(self, *args):
        self.rect.pos = self.pos
        self.rect.size = self.size
```

或者也可以清空Canvas画布然后重新画：

```
class MyWidget(Widget):
    def __init__(self, **kwargs):
        super(MyWidget, self).__init__(**kwargs)
        self.draw_my_stuff()

        self.bind(pos=self.draw_my_stuff)
        self.bind(size=self.draw_my_stuff)

    def draw_my_stuff(self):
        self.canvas.clear()

        with self.canvas:
            self.rect = Rectangle(pos=self.pos, size=self.size)
```

要注意更新指令的方法是更好的选择，因为这样减少了开销，并且避免了创建新指令。

Title: Kivy Pack Windows Date: 2017-03-01 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：打包为 Windows 系统可执行文件

英文原文

特别注意

本文档仅适用于 `1.9.1` 以及更新版本的 Kivy。

要打包 Windows 平台的应用程序，只能在 **Windows** 操作系统下完成。另外一定要注意，本文后续内容中都是在 **wheels** 安装的 Kivy 下通过测试的，如果你用其他安装方法，参考结尾部分吧。

打包出来的程序是 32 位还是 64 位只取决于你打包使用的 Python，而不取决于 Windows 操作系统的版本。

依赖包

- 最新版本的Kivy （参考安装指南 Installation on Windows）
- PyInstaller 3.1或者更新的版本 (`pip install --upgrade pyinstaller`)。

（译者注：PyInstaller 目前（2017年03月01日）还不支持 Python 3.6 哦~，好多朋友都坑到这里了，所以推荐使用 3.5.2。）

PyInstaller 的基本用法

本节内容是要让 PyInstaller（3.1或者更新版本）包含 Kivy 的 Hook（钩子， Windows 消息处理机制的一个平台）。要覆盖默认的 Hook ，下面的样例代码需要稍微修改一下。参考 覆盖默认 Hook。

打包一个简单的 APP

这个例子里面，咱们要把样例中的 **touchtracer** 这个项目进行打包，并且添加一个自定义图标。这里 Kivy 的样例目录要注意，如果是用 wheels 安装的，在 `python\\share\\kivy-examples` 这个位置，如果从 github 上面下载的，就在 `kivy\\examples` 这个位置。为了避免

混乱，这里就用 `examples-path` 来指代这个目录的完整路径。然后 touchtracer 这个样例在 `examples-path\\demo\\touchtracer` 这个文件夹里，代码文件是 `main.py` 。

1 确保 Python

打开命令行确保 Python 包含在环境变量内，也就是说，输入 `python` 会出现解释器提示符。（译者注：cmd 或者 powershell 都可以，更推荐用后者，语法和 Bash 比较相似。）

2 创建文件夹

在要打包 APP 的位置创建一个文件夹。比如咱们这次就创建一个名字为 `TouchApp` 的文件夹，然后用类似 `cd TouchApp` 这样的命令进入到这个新建目录内。之后输入：

```
python -m PyInstaller --name touchtracer examples-path\demo\touchtracer\main.py
```

还可以增加一个 icon.ico 文件到这个应用目录，这样就可以让程序有自己的图标了。如果没有自己的 .ico 图标文件，可以把你的 icon.png 文件转换成 ico，用这个 ConvertICO 在线的应用就可以了。保存 icon.ico 到 touchtracer 这个目录里面，然后输入：

```
python -m PyInstaller --name touchtracer --icon examples-path\demo\touchtracer\icon.ico examples-path\demo\touchtracer\main.py
```

更多其它选项，请参考 PyInstaller 官方说明。

3 编辑配置文件

在 `TouchApp` 里面会有一个配置文件 `touchtracer.spec` 。咱们需要编辑修改一下这个文件，在里面增加一些依赖包的 hook，这样才能保证正确创建 exe。接下来就是打开编辑器了，爱用啥都行，然后在配置文件的开头添加上下面这句：（这里是假设用的是 sdl2， 现在 Kivy 默认使用这个）

```
from kivy.deps import sdl2, glew
```

然后，用搜索，找到 `COLLECT()` 这个位置，添加上 touchtracer 用到的其他文件（touchtracer.kv, particle.png,等等）：修改示例中的行位置，添加一个 `Tree()` 对象，例如这里的是 `Tree('examples-path\\demo\\touchtracer\\')` 。这个 `Tree()` 会搜索在当前这个 touchtracer 文件夹的所有文件，并添加到你最终打包的程序中。

要添加额外的依赖包，就要在 COLLECT 的第一个关键词参数的前面，为每一个依赖包的路径添加一个 Tree 对象。例如下面的就是以 `*[Tree(p) for p in(sdl2.dep_bins + glew.dep_bins)]` 为例：

```
coll = COLLECT(exe, Tree('examples-path\\demo\\touchtracer\\'),
                   a.binaries,
                   a.zipfiles,
                   a.datas,
                   *[Tree(p) for p in (sdl2.dep_bins + glew.dep_bins)],
                   strip=False,
                   upx=True,
                   name='touchtracer')
```

4 进行构建

接下来就用 `TouchApp` 里面这个 spec 配置文件来进行构建了：

```
python -m PyInstaller touchtracer.spec
```

5 生成位置

编译好的包会在 TouchApp\dist\touchtracer 这个目录。

使用 gstreamer 创建一个视频应用

接下来就是修改一下上面的这个样例了，这回要打包的 APP 是一个使用了 gstreamer 的视频应用。咱们这回用样例中的视频播放器的例子 `videoplayer`，代码在 `examples-path\widgets\videoplayer.py`。另外创建一个名字为 `VideoPlayer` 的文件夹，然后在命令行中进入到这个文件夹，之后操作如下：

```
python -m PyInstaller --name gstvideo examples-path\widgets\videoplayer.py
```

这回要修改 `gstvideo.spec` 这个文件。跟上文的方法类似，也就是把 gstreamer 的依赖包放进去：

```
from kivy.deps import sdl2, glew, gstreamer
```

然后就是增加 `Tree()` 来包含好要用的视频文件，`Tree('examples-path\\widgets')` 和 gstreamer 依赖都得弄好，大概如下所示：

```
coll = COLLECT(exe, Tree('examples-path\\widgets'),
               a.binaries,
               a.zipfiles,
               a.datas,
               *[Tree(p) for p in (sdl2.dep_bins + glew.dep_bins + gstreamer.dep_bins)
],
               strip=False,
               upx=True,
               name='gstvideo')
```

接下来就是使用 `VideoPlayer` 文件夹中的这个 spec 配置文件来进行构建了：

```
python -m PyInstaller gstvideo.spec
```

然后你就能在 `VideoPlayer\dist\gstvideo` 这个位置找到 gstvideo.exe 这个文件了，运行一下就能播放视频了。

特别注意

如果你用了 Pygame，或者你打包的程序需要 Pygame，那在你的 spec 文件里面就还得添加如下的代码，在 import 导入语句的后面添加（详情参考 kivy issue #1638）：

```
def getResource(identifier, *args, **kwargs):
    if identifier == 'pygame_icon.tiff':
        raise IOError()
    return _original_getResource(identifier, *args, **kwargs)

import pygame.pkgdata
_original_getResource = pygame.pkgdata.getResource
pygame.pkgdata.getResource = getResource
```

覆盖默认 Hook

包含/移除视频音频以及缩小应用体积

PyInstallers 默认会将 Kivy 用到的所有核心模块和这些模块的依赖包，全部都添加成 hook，比如音频，视频，拼写等等（然而 gstreamer 的 dll 还是需要你用 `Tree()` 来手动添加，参考上文）。有的 Hook 并没有用到或者想要缩小应用的体积，就都可以尝试着移除一些模块，比如如果没有用到音频和视频，就可以用自定义的 Hook了。

Kivy 在 `hookspath()` 提供了可选的 Hook。

此外，当且仅当 PyInstaller 没有默认的 hook 的时候，就必须得提供一个 `runtime_hooks()` 。覆盖 hook的时候，这个 `runtime_hooks()` 不需要覆盖。

可选自定义的 `hookspath()` hook不包含任何 Kivy 的 provider。要添加电话，就要用 `get_deps_minimal()` 或者 `get_deps_all()` 来添加。可以看看相关的文档以及 `pyinstaller_hooks` 来了解更多信息。不过 `get_deps_all()` 跟默认的 hook一样，都是把所有 provider 都添加进去；而 `get_deps_minimal()` 只添加在应用程序运行的时候加载了的内容。

这两个方法都提供了一个 Kivy 隐藏导入列表，以及排除的导入，可以传递出来给 `Analysis` 。

还可以生成一个自定义 hook，一个个列出每一个 kivy 的 provider 模块，然后把其中用不上的就注释掉就行了。

参考 `pyinstaller_hooks` .

要在上面的例子中使用自定义 hook，要按照下面给出的范例来修改，以 `hookspath()` 和 `runtime_hooks` （必要情况下），然后是 `**get_deps_minimal()` 或者 `**get_deps_all()` 来制定好各种 provider。

例如，增加了导入语句 `from kivy.tools.packaging.pyinstaller_hooks import get_deps_minimal, get_deps_all, hookspath,runtime_hooks` ，然后按照如下方式修改 `Analysis` ：

```
a = Analysis(['examples-path\\demo\\touchtracer\\main.py'],
             ...
             hookspath=hookspath(),
             runtime_hooks=runtime_hooks(),
             ...
             **get_deps_all())
```

上面这个实际上跟默认 hook 一样包含全部了。或者可以：

```
a = Analysis(['examples-path\\demo\\touchtracer\\main.py'],
             ...
             hookspath=hookspath(),
             runtime_hooks=runtime_hooks(),
             ...
             **get_deps_minimal(video=None, audio=None))
```

这样就是移除了声音视频的 provider，这就只加载了用到的核心模块了。

关键就是要提供自定义的 `hookspath()` ，这个默认并不会列出全部的 kivy provider，而是手动的来设定隐藏导入的模块和需要用的 provider，通过 `get_deps_minimal()` 来移除用不上的模块 (比如上面的是声音影像)。

其他安装方式

前面的这些例子都用到了 `*[Tree(p) for p in (sdl2.dep_bins + glew.dep_bins + gstreamer.dep_bins)],` 这样的语句来保证 PyInstaller 把所有依赖包用到的 dll 都添加进去。如果不是用 wheel 的方法安装的 Kivy，那么这些命令就很可能失败，比如 `kivy.deps.sdl2` 可能就无法导入。这时候，你就必须得找到这些 dll 文件的位置，然后手动地一个个传递给 `Tree` 类，传递方法和上面说的基本差不多了。

（译者注：Windows 平台还是推荐用 wheel 二进制安装，省心多了。）

Title: Kivy Android Date: 2017-03-07 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：KV Android 详细指南

英文原文

在 Android 设备上，只要是支持 OpenGL ES 2.0 （至少为 Android 2.2 以及其后的版本），就基本都能运行 Kivy 应用程序。OpenGL ES 2.0 基本上是现代设备的标准了；根据谷歌的报道说，至少有 99.9% 的设备都是支持的。

Kivy 的 APK 就是常规的 Android APP，和其他的 APP 一样可以四处发布，比如谷歌 Play 商店等等。在暂停或者重启的时候这些应用的行为也都很正常，下面要介绍道的是 Kivy APP 用到的 Android 服务和需要用到的大多数常规的 Java API。

下面的内容依次讲解了如何 针对 Android 平台打包 APP， 在设备上调试 APP，以及使用震动或者读取传感器等等 Android API。

针对 Android 平台打包 APP¶

Kivy 项目提供了针对 Android 平台打包 APP 所需的全部必备工具，可以构建单个的 APK 文件发布到谷歌 Play 市场之类的应用商店。详细内容可以参考针对 Android 打包应用程序的英文原版文档或者中文翻译版本：个人博客地址，知乎专栏地址。

在 Android 设备上调试 APP¶

通过 Android Logcat 流，可以观察代码的常规输出（比如 stdout，stderr），也可以查看常规的 Kivy 日志。这需要使用 adb 来查看，adb 包含在 Android SDK 内。这首先就要求你再设备上开启开发者模式，然后启用 USB 调试功能来启用 adb，接着把设备连接到计算机，在终端中运行下面的命令：

```
adb logcat
```

这样就能看到日志输出了，包括标准输出和出错信息（ stdout/stderr ）以及 Kivy 的日志。

如果你用 Buildozer 打包的 APP，那么有可能 adb 工具没有包含在你的 `$PATH` 环境变量中，这样上面的命令就可能没有效果。这时候可以用下面的方法：

```
buildozer android logcat
```

上面这样就是运行了 Buildozer 伴随安装的 adb 工具，或者还可以去 `$HOME/.buildozer/android/platform` 这个目录找到 Buildozer 伴随安装的 SDK。

或者还可以下载Kivy Launcher来运行和调试应用程序。如果用这种方法运行 Kivy 应用程序，可以在你的应用程序所在目录下找到一个名字为 `/.kivy/logs` 的子目录，里面就是日志文件。

使用 Android API¶

虽然 Kivy 是一个 Python 框架，Kivy 项目也还维护了一套用来调用常规 Java API 的工具，可以用来处理震动、传感器、发送短信或邮件等等。

对新用户来说，推荐阅读 Plyer。要想有更深入的使用或者调用一些目前没有封装的 API，可以直接使用 Pyjnius。Kivy 还内置了一个 Android 模块 来实现 Android 的一些基础功能。

在 Kivy wiki 上可以找到用户提供的 Android 代码和样例。

Plyer¶

Plyer 是一个 Python 风格的，独立于平台的 API，用于使用各种平台上都普遍具有的功能，尤其是移动平台。其设计思路是你的应用可以简单地调用一个 Plyer 函数，例如给用户一个消息通知，然后 Ply会处理在不同的平台或者操作系统下分别如何把这件事完成。在 Plyer 的内部，在 Android 平台使用的是 Pyjnius，在 iOS 平台使用了 Pyobjus，在桌面平台又用了其他的特定 API。

例如，下面的代码就会让你的 Android 设备震动，或者当你在其他平台不恰当地使用的时候就会跑出一个 NotImplementError，例如在桌面平台等没有对应硬件的情况下：

```
from plyer import vibrator
vibrator.vibrate(10)  # vibrate for 10 seconds
```

Plyer 支持的 API 越来越多了，可以在 Plyer GitHub 页面的 README 文件中查看完整的支持列表。

Pyjnius¶

Pyjnius 是一个 Python 模块，它允许你直接在 Python 中读取 Java 类，自动转换参数成正确的类型，还允许你把 Java 的运行结果转换给 Python。

Pyjnius 可以从 它的 GitHub 地址下载获得，并且有一份详细的文档。

下面的代码是一个简单的小例子，展示了如何使用 Pyjnius 来读取常规的 Android 震动 API，就跟上面 Plyer 的代码效果一样：

```
# 'autoclass' 接收一个Java 类，然后打包给 Python；
from jnius import autoclass

# Context 是 Android API 中一个常用的 Java 类；
Context = autoclass('android.content.Context')

# PythonActivity 是在 python-for-android 内由 Kivy bootstrap app 提供的一个类；
PythonActivity = autoclass('org.renpy.android.PythonActivity')

# 这里的 PythonActivity 存储了一个指向当前运行的 Activity 的引用；
# 我们要用它来读取 震动服务
activity = PythonActivity.mActivity

# 底下的这个振动器代码和 Java 里面基本一样的；
vibrator = activity.getSystemService(Context.VIBRATOR_SERVICE)

vibrator.vibrate(10000)  # 这个值是毫秒为单位，这里设置的 10 000 毫秒相当于 10 秒。
```

上面的代码直接用了 Java API 函数来调用了振动器，Pyjnius 自动把 API 转换出给了 Python 代码使用，而又把我们的调用都回传给了 Java。相比 Plyer 的实现，这种方法更繁琐一些，也更像 Java 的风格，在这个例子中没有什么优势。不过 Plyer 也并没有对 Pyjnius 的所有 API 都进行了封装。

Pyjnius 还有一个强大的功能就是实现 Java 接口，这在封装某些 API 的时候非常重要，不过这里就不详细讲这么多了，有兴趣的话去 Pyjnius 的官方文档来了解更深层次内容吧。

Android 模块¶

Python-for-android 项目中包含了一个 Python 模块（实际上是用 cython 封装的 Java）来读取一系列有限的 Android API。这个很大程度上已经被上面的 Pyjnius 和 Pyler 取代了，因为后者更加灵活方便，不过有时候可能这个模块还有些用处。所有可用的文档都可以在 python-for-android 官方文档中查阅到。

这其中就包括了计费/应用内购买的代码，以及创建或者读取某些 Android 服务的代码，其他工具目前还没能提供这方面的功能。

项目状态以及通过测试的设备¶

前面的章节讲述了 Kivy 在 Android 系统的构建工具，以及他们各自的缺陷还有就是已知能够使用的设备。

Android 工具链现在挺稳定的，一定程度上基本能适用于各种设备了；Kivy 的最低要求是 OpenGL ES 2.0 以及 Android 2.2。这现在绝对是覆盖面很广泛了— Kivy 已经都可以在 Android 智能手表上面运行了。

当前在技术上存在的一个限制就是 Android 构建工具只能生成 ARM 平台的 APK 文件，这些文件不能运行于 X86 处理器的 Android 设备上，好在目前 这类 X86 的 Android 设备还不是主流。不过对 X86 处理器的 Android 设备的支持是后续要添加的。

因为目前 Kivy 基本上能在绝大多数 Android 设备上面良好运行了，所以之前的那个设备支持列表就关荣退休了—只要满足上面的要求的 Android 设备，基本就都能够使用。

Title: Kivy Pack Android Date: 2017-03-06 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：打包为 Android 系统可执行文件

英文原文

你可以通过 python-for-android 这个项目来打包一个 Android 应用。本页面详细讲解如何下载和打包，可以在你自己的机器上直接进行（参考此页面），或者使用预先构建好的Kivy Android 虚拟机，或者使用Buildozer 来自动化完成整个过程。还可以参考 针对 Kivy Launcher 进行打包 这样就不用编译就能运行 Kivy 应用。

对新手，Kivy 官方推荐使用 Buildozer ，这是制作完整 APK 的最简单的途径。或者也可以使用 Kivy Launcher 这个应用来运行你的 Kivy 应用，而不用编译了。

Kivy 应用可以发布到 Android 应用市场，比如谷歌的 Play 市场等等，只需要额外几步来创建一个完整签名的 APK 就可以了。

Kivy 项目包含了一系列读取 Android API 的工具，可以实现震动、传感器读取、信息发送等等功能。相关的详细信息都可以参考 Kivy 的 Android 专题页面。

特别注意

Android 平台目前已经支持 Python 3 了，不过还处于实验阶段。

Buildozer¶

Buildozer 是一个将整个构建过程自动化的工具。它会下载和设置 python-for-android 需要的所有依赖项目，包括 Android 的 SDK 和 NDK，然后会构建 APK ，这个 APK 可以自动推送到设备上。

目前 Buildozer 只能用在 Linux 上面，而且还不是正式版，处于测试阶段，发布的是 alpha 版本，不过目前用起来还不错，能显著简化 APK 构建的过程。

可以到 Buildozer 的项目页面 下载获取 Buildozer。

```
git clone https://github.com/kivy/buildozer.git
cd buildozer
sudo python2.7 setup.py install
```

上面的命令就会把 Buildozer 安装到你的操作系统中。接下来就是到你的项目目录然后运行如下命令：

```
buildozer init
```

这会在你的目录下创建一个名为 buildozer.spec 的文件，这个文件是控制项目构建选项的。估计你需要编辑修改一下这个文件，比如设置你应用的名字等等。在这里可以设置传递给 python-for-android 的全部或者大部分参数。

安装 Buildozer 的依赖项目。

最后一步了，连接上你的 Android 设备然后运行下面的命令：

```
buildozer android debug deploy run
```

这样就可以创建、推送 APK 到你的设备上，然后就可以自动运行了。

Buildozer 有很多可以控制的选项和工具，对你都会游泳，上面这些步骤只是创建 APK 的最简单的方法。可以到 Buildozer 的官方文档页面查看完整说明。也可以看看 Buildozer 项目页面的 README 文件。

通过 python-for-android 打包¶

你还可以直接用 python-for-android 来打包应用，这样你可以有更多控制选项，但需要手动下载和设置 Android 工具链。

参考python-for-android 官方文档 查看全部细节。

针对 Kivy Launcher 打包¶

Kivy launcher 是一个 Android 应用，可以运行存储到 SD 卡里面的 Kivy 样例。可以用下面的方法来安装 Kivy launcher：

1. 前往谷歌 Play 市场中的 Kivy Launcher 页面；
2. 点击安装；
3. 选择你的设备，然后就搞定了。

如果你的设备无法访问谷歌 Play 市场（我大天朝么。。。），可以从 Kivy 官网的下载页面手动下载安装 APK 文件。

安装了 Kivy launcher 之后，就可以把你的 Kivy 应用放到外置存储的 Kivy 文件夹中，（通常是在 `/sdcard` 目录下），例如：

```
/sdcard/kivy/yourapplication>
```

`<yourapplication>` 你的应用应该是一个文件夹，包含以下文件：

```
# 程序主文件：
main.py
# Kivy 需要的关于你应用的信息：
android.txt
```

以下信息是 android.txt 这个文件必须包含的：

```
title=<Application Title> #这是应用的标题
    author=<Your Name>  #这是作者签名
orientation=<portrait|landscape> #设定屏幕方向为水平或竖直
```

上面这些都是非常基础的设置。如果你要用上面的工具来构建自己的 APK，还得调整很多其他的设定。

安装样例项目¶

Kivy 自带了很多歌样例，可以先用这些来试一试 Kivy Launcher。可以用如下的方法来运行：

1. 下载 `Kivy demos for Android`
2. 解压缩然后进入目录 `kivydemo-for-android`
3. 把目录内所有子目录等内容复制到 `/sdcard/kivy` 这个目录
4. 运行 Kivy Launcher 然后从样例中选择一个来试试，Pictures, Showcase, Touchtracer, Cymunk 等等都可以。

发布到应用市场¶

如果你用 Buildozer 或者 python-for-android 构建了 APK 文件，就可以创建一个 release 版本来发布到谷歌 Play 市场或者其他 Android 应用市场。

要想发布，就必须在运行 Buildozer 的时候添加上 `release` 参数，例如 `buildozer android release` ，如果用了 python-for-android 就在 build.py 进行编译的时候加上 `--release` 。

这样就能在 `bin` 目录里面创建一个正式发布的 release 版的 APK 文件，需要你做好签名和 zipalign 压缩优化（译者注：zipalign，优化apk应用程序的工具，使包内未压缩的数据能够有序的排列）。

上面这些操作的详细过程可以参考 Android 官方文档 ，所有用到的工具都在 Android SDK 里面了。

设定 Android ¶

Kivy 的设计定位是提供跨平台的相同操作体验，因此也有一些清晰地设计特点。Kivy 包含了自己独有的一套控件，在默认情况下，用所有需要的依赖包和链接库来构建 APK 文件。

也可以指定特定的 Android 功能，可以直接进行，也可以用（某些）跨平台的方式来实现。更多细节可以参考 Kivy 的 Android 专题页面 中关于使用 Android API 的部分。

Title: Kivy VM Date: 2017-03-6 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：Kivy 虚拟机

英文原文

简介¶

目前，Kivy 的 Android 应用程序构建只能在配置有 python-for-android、 Android SDK 和 NDK 的 Linux 环境进行。这对于 Windows 或者 OS X 的用户来说，搭建起来就有点麻烦，所以 Kivy 官方提供了一个完全配置好的 VirtualBox 虚拟机磁盘镜像，用来减轻自己搭建的哀伤与痛苦。

如果你对于虚拟机不太了解，建议去阅读以下维基百科上面的虚拟机页面。

上手¶

1. 首先是到 Kivy 下载页面 找到 **Virtual Machine** 这一部分。下载的文件超过 2GB ，解压缩之后是大概 6GB。解压缩文件之后别忘掉 vdi 虚拟磁盘的位置。
2. 根据你机器的操作系统版本，去 VirtualBox 下载页面 下载对应的安装包，然后安装。
3. 启动 VirtualBox，点击新建按钮。然后选择操作系统为 “linux” ，版本设置为 “Ubuntu 64-bit”。
4. 在 “Hard drive” 硬盘这个选项，选择“Use an existing virtual hard drive file”，即使用已有的虚拟硬盘文件。这时候找到上面你下载并解压出来的那个 vdi 文件，选中使用它。
5. 到虚拟机设置的页面。在“Display -> Video” 显示器->显卡，这一部分，吧显存增加到至少 32MB 以上。启用 3D 加速来提高用户体验。
6. 启动虚拟机，然后看看桌面上的 readme 文件，根据指示操作就行了。

构建 APK¶

虚拟机家在之后，就根据 python-for-android 打包指南里面的讲解来构建 APK 文件里了：英文原版，中文翻译版本或者知乎专栏的镜像。也根本不用使用 git clone 下载什么的了，因为虚拟机里面的 python-for-android 已经安装配置好了，就在虚拟系统的 home 目录里面了。

提示建议¶

1. 共享文件夹 通常情况下，你的开发环境和工具链都在宿主机中，而 APK 的构建在客户机

里面。好在 VirtualBox 提供了共享文件夹的功能，允许你的客户机直接读取宿主机中的某个文件夹。 可以选中‘Permanent’（永久挂载）和 ‘Auto-mount’（自动挂载）这两个选项，这样把构建好的 APK 文件复制到宿主机就更方便了。写一个小脚本就可以很简单地实现自动复制或者移动这个步骤。

2. 复制粘贴 默认情况下，宿主机和客户机的剪贴板是不能共享的。可以在“Settings -> General -> Advanced”（设置->通用->高级）中启用 “bi-directional”（双向复制粘贴）的选项。
3. 虚拟机快照 如果你正在用 Kivy 开发环境的分支，同步最新版本有时候可能会出问题（Kivy 开发者尽量在避免这种情况）。所以可以在更新之前建立一个虚拟机快照来避免这类问题。这能让你很方便恢复到之前能用的状态。
4. 补充内存 如果虚拟机分配的内存不够，可能会因为一些很神奇的错误导致编译失败，比如：

```
arm-linux-androideabi-gcc: Internal error: Killed (program cc1)
```

如果出现了上面这种情况，检查一下 Kivy 虚拟机的剩余内存，如果内存不足，就在虚拟机设置里面多增加一些吧。

Title: Kivy Pack Mac Date: 2017-03-02 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：打包为 Mac 系统可执行文件

英文原文

特别注意

本文所提供的打包 Kivy 应用程序的方法必须在 OS X 系统内进行，而且不再支持 32 位平台。

使用 Buildozer

```
pip install git+http://github.com/kivy/buildozer cd /to/where/I/Want/to/package buildozer init
```

（译者注：这里的/to/where/I/Want/to/package 就是你要打包的应用所在目录。）

然后就还是修改 buildozer.spec 文件，在里面添加好你的应用需要用到的信息。添加依赖包的位置在 requirements= 的那个位置。

默认情况下，requirements 位置所指定的 Kivy 版本会被忽略掉。

如果你在应用程序目录下有 Kivy.app （/Applications/Kivy.app），那就会用这个来打包。如果没有，还可以从 Kivy.org 下载最新版来用了。

如果你要用 Python3 来打包，就就直接从 Kivy.org 的下载页面下载 Kivy3.7z 这个包，然后把它解压缩到应用目录 /Applications 下，命名为 Kivy.app，然后运行：

```
buildozer osx debug
```

打包好应用之后，就可以移除用不上的包了，比如如果你不用视频功能，那就可以去掉 gstreamer。同理，其他的用不上的功能，也都可以去掉，这就可以保证打包出来的应用能够有尽量小的体积，足够运行就好了。

作为示例，我们用这个方法打包好了一个应用，Python2 的版本大概 9MB 多一点，Python3 的是 15MB 左右，可以在官方提供的谷歌网盘下载来体验一下。（译者注：如果你身处大陆而无法访问谷歌这种不存在的网站，那么可以试试我的百度网盘分享。）

就这么多，动手试试吧。

目前的 Buildozer 使用了 Kivy SDK 来打包你的应用程序。如果你想对你的程序进行更深入的修改定制，而 buildozer 不足以满足你的需求，你可以试试直接使用 SDK，下面就要详细介绍一下这部分。

使用 Kivy SDK

从 1.9.0 版本开始，Kivy 就开始发布针对 OS X 平台的自我包含的便捷包。

用下面描述的方法就可以使用 Kivy SDK 来打包和发布应用程序了，要添加一些诸如 SDL2 或者 GStreamer 之类的包也都更简单了。

1 首先要确保有未修改过的原版 Kivy SDK，也就是从下载页面获取的 Kivy.app 这个文件。

2 然后运行下面的命令：

```
mkdir packaging
cd packaging
packaging> git clone https://github.com/kivy/kivy-sdk-packager
packaging> cd kivy-sdk-packager/osx
osx> cp -a /Applications/Kivy.app ./Kivy.App
```

特别注意

上面这一步是至关重要的，一定要确保目录和权限都没有问题。`cp -rf` 这样的命令也能实现复制，但会让应用程序无法运行，并且在后续步骤中导致各种错误。

3 接下来就是要把你用 Kivy.app 编译好的应用程序包含进目标文件夹，使用如下命令：

```
osx> ./package-app.sh /path/to/your/app_folder_name>/
```

就是你应用程序的名字。

这个命令会把 Kivy.app 复制成 .app，并把应用程序的一份编译好的副本包含进去。

4 就这些了，你的应用程序已经打包完毕，可以拿出去安装了。接下来可以按照下面的方法对你的应用进行更进一步的定制了。

安装模块

OS X 上的 Kivy 包邮自己的虚拟环境，当你使用 kivy 命令来运行的时候就会激活这个虚拟环境。如果要安装额外的一些模块，可以用如下命令：

```
kivy -m pip install
```

模块和文件安到哪了？

在 Kivy.app 这个文件内部，虚拟环境位置如下：

```
Kivy.app/Contents/Resources/venv/
```

如果你安装了一个安装二进制的模块，比如 kivy-garden。那么这些二进制文件只能在虚拟环境中才是可用的，就比如你先运行了下面这个命令：

```
kivy -m pip install kivy-garden
```

然后安装的 garden lib 就只在你激活这个虚拟环境的时候才可用。

```
source /Applications/Kivy.app/Contents/Resources/venv/bin/activate garden install mapv
iew deactivate
```

安装二进制文件

这个比较简单，就把二进制文件复制到虚拟目录下的 bin 文件夹就可以了。（Kivy.app/Contents/Resources/venv/bin/）

包含其他框架

Kivy.app 已经自带了 SDL2 和 Gstreamer 这两个框架。要包含其他框架可以参考下面的方法：

```
git clone http://github.com/tito/osxrelocator
export PYTHONPATH=~/path/to/osxrelocator
cd Kivy.app
python -m osxrelocator -r . /Library/Frameworks/<Framework_name>.framework/ \
@executable_path/../Frameworks/<Framework_name>.framework/
```

Do not forget to replace with your framework. This tool osxrelocator essentially changes the path for the libs in the framework such that they are relative to the executable within the .app, making the Framework portable with the .app.

一定别忘了把上面样例中的 替换成你要安装的框架名字。osxrelocator 这个工具可以改变框架中的链接库的路径，这样就能让它们指向 .app 文件内的可执行文件，也就让此框架成为 .app 文件内可用的内置框架了。

缩小应用体积

现在这个应用程序的体积可能已经相当大了，好在很多没有用上的部分可以从包中移除。

举例来说，如果你没有使用 Gstreamer，就可以从你应用程序的 .app 文件内的 /Contents/Frameworks 目录中把它删除掉。类似的像是在 /Applications/Kivy.app/Contents/Resources/kivy/ 目录下的 examples，tools，docs 等等这些文件夹都可以删掉的。

这样就可以让你的包只包含你的应用程序用到的内容。

修改设置

通过修改应用程序的 .app 文件内 /Contents/info.plist 这个文件，就可以修改图标和其他的设置了。

创建 DMG

用如下命令就可以创建一个 DMG 镜像文件了：

```
osx> ./create-osx-dmg.sh YourApp.app
```

一定要注意末尾没有额外的 /。 这样就能生成一个压缩的 DMG 文件，能进一步缩小应用发布时候的体积了。

使用 PyInstaller，无 HomeBrew

首先是安装 Kivy 和依赖包，不用 HomeBrew 的方法可以在官方文档 或者译者的博客或者译者的专栏中查找。

安装好了Kivy 以及依赖包之后，就需要安装 PyInstaller 了。

（译者注：PyInstaller 目前不支持 Python3.6，时间为2017-03-02。）

假设用一个名为 testpackaging 的文件夹：

```
cd testpackaging
git clone http://github.com/pyinstaller/pyinstaller
```

在这个目录中创建一个名为 touchtracer.spec 的配置文件，然后添加如下的代码到该文件中：

```
# -*- mode: python -*-

block_cipher = None
from kivy.tools.packaging.pyinstaller_hooks import get_deps_all, hookspath, runtime_ho
oks

a = Analysis(['/path/to/yout/folder/containing/examples/demo/touchtracer/main.py'],
             pathex=['/path/to/yout/folder/containing/testpackaging'],
             binaries=None,
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             hookspath=hookspath(),
             runtime_hooks=runtime_hooks(),
             **get_deps_all())
pyz = PYZ(a.pure, a.zipped_data,
             cipher=block_cipher)
exe = EXE(pyz,
          a.scripts,
          exclude_binaries=True,
          name='touchtracer',
          debug=False,
          strip=False,
          upx=True,
          console=False )
coll = COLLECT(exe, Tree('../kivy/examples/demo/touchtracer/'),
               Tree('/Library/Frameworks/SDL2_ttf.framework/Versions/A/Frameworks/Free
Type.framework'),
               a.binaries,
               a.zipfiles,
               a.datas,
               strip=False,
               upx=True,
               name='touchtracer')
app = BUNDLE(coll,
             name='touchtracer.app',
             icon=None,
         bundle_identifier=None)
```

把路径改到你的相对路径：

```
a = Analysis(['/path/to/yout/folder/containing/examples/demo/touchtracer/main.py'],
            pathex=['/path/to/yout/folder/containing/testpackaging'],
...
...
coll = COLLECT(exe, Tree('../kivy/examples/demo/touchtracer/'),
```

然后运行如下命令：

```
pyinstaller/pyinstaller.py touchtracer.spec
```

把这里的 touchtracer 替换成你的应用名称。之后就可以在 dist 文件夹下看到你的 .app 文件了。

使用 PyInstaller + HomeBrew

特别注意

打包你的应用程序的时候，你定要在你要兼容的最老版本的 OS X 系统上进行。

完整指南

1 安装 Homebrew

2 安装 Python:

```
brew install python
```

特别注意

如果要用 Python3 ，就用 `brew install python3` ，然后把下文中的 `pip` 改成 `pip3` 就可以了。

3 (Re)install your dependencies with `--build-bottle` to make sure they can be used on other machines:

```
brew reinstall --build-bottle sdl2 sdl2_image sdl2_ttf sdl2_mixer
```

特别注意

如果你的项目依赖 Gstreamer 或者其他的链接库，一定要按照下文的方法添加 `--build-bottle` 来安装。

4 安装 Cython 和 Kivy:

```
pip install -I Cython==0.23
USE_OSX_FRAMEWORKS=0 pip install -U kivy
```

5 安装 PyInstaller:

```
pip install -U pyinstaller
```

6 使用到 main.py 的路径来打包应用：

```
pyinstaller -y --clean --windowed --name touchtracer \
      --exclude-module _tkinter \
      --exclude-module Tkinter \
      --exclude-module enchant \
      --exclude-module twisted \
      /usr/local/share/kivy-examples/demo/touchtracer/main.py


#### 特别注意

这样不能把额外的图像和声音文件复制进去。要添加这些内容还是要创建一个专门的`.spec`配置文件。

### 编辑`.spec` 配置文件

咱们用的这个配置文件是 touchtracer.spec ， 位置在刚刚运行了 pyinstaller 的目录。

需要对配置文件中的 COLLECT() 调用的部分进行修改，要添加上 touchtracer 用到的资源（比如touchtrace
r.kv， particle.png，等等）。修改这一行，添加一个 Tree() 对象。这个 Tree 会搜索 touchtracer 目
录下的所有文件，并添加到你的包当中。 COLLECT 的那部分代码应该大概如下所示：


```Bash
coll = COLLECT(exe, Tree('/usr/local/share/kivy-examples/demo/touchtracer/'),
               a.binaries,
               a.zipfiles,
               a.datas,
               strip=None,
               upx=True,
               name='touchtracer')
```

这样会把需要的文件都添加进去，这样 PyInstaller 就能包含需要用到的 Kivy 文件了。弄妥了之后，你的 spec 配置文件就可以执行了。

使用 spec 来构建并打包成 DMG

1 打开终端。

2 进入到 PyInstaller 的目录，然后用如下命令进行构建：

```
pyinstaller -y --clean --windowed touchtracer.spec
```

3 运行来试试：
```

```
pushd dist
hdiutil create ./Touchtracer.dmg -srcfolder touchtracer.app -ov
popd
```

4 然后就能在 dist 目录下找到 Touchtracer.dmg 这个文件了。

额外的链接库

GStreamer

如果你的项目需要 GStreamer，那就运行如下命令：

```
brew reinstall --build-bottle gstreamer gst-plugins-{base,good,bad,ugly}
```

特别注意

如果你的项目需要对 Ogg Vorbis（音频压缩格式，类似于MP3，完全免费、开放和没有专利限制，支持多声道） 的支持，一定要在上面的命令后加上 `--with-libvorbis` 。

如果你用的是通过 HomeBrew 安装的 Python，那就还需要下面这一步，除非这个问题弄妥了：

```
brew reinstall --with-python --build-bottle https://github.com/cbenhagen/homebrew/raw/
patch-3/Library/Formula/gst-python.rb
```

Title: Kivy Pack iOS Date: 2017-03-07 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：打包为 iOS 系统可执行文件

英文原文

特别注意

目前还只能用 Python 2.7 来针对 iOS 平台打包应用程序。 Python 3.3 以上的支持还在开发中。

`The overall process for creating a package for IOS can be explained in 4 steps:`

总体上创建一个 iOS 应用程序需要四步：（译者注：对，你没有看错，官方文档就是写的 4 steps，所以我特地留下了上面这行原文，可见估计 Kivy 的开发者们也没太注意这点小错误吧。。。）

1. 针对 iOS 编译 Python 和 需要的模块；
2. 创建一个 Xcode 项目，链接你的源代码；
3. 进行定制修改。

预先准备¶

首先要安装一些依赖包，比如 cython，autotools 等等。Kivy 的开发者推荐你使用 Homebrew 来安装这些依赖包：

```
brew install autoconf automake libtool pkg-config
brew link libtool
sudo easy_install pip
sudo pip install cython==0.23
```

更多细节参考 IOS Prerequisites。一定要在下一步开始之前保证依赖关系都满足了。

编译发布版¶

Open a terminal, and type:

打开终端，输入下面的命令：

```
git clone git://github.com/kivy/kivy-ios
cd kivy-ios
./toolchain.py build kivy
```

Python 发型版中大部分内容都会被打包到 python27.zip 这个文件中。如果遇到了问题，可以参考谷歌论坛用户组 或者 kivy-ios 项目页面。

创建一个 Xcode 项目¶

在进行下一步之前，要保证你的程序运行起点是一个名为 main.py 的文件。

Kivy 官方提供了一个脚本，可以创建一个初始的 Xcode 项目。在下面的代码样例中，把 Touchtracer 这个换成你的项目名字。名字一定不能有空格或者其他非法字符。（译者注：吐槽一下，那什么算是非法字符倒是说说啊。）

```
./toolchain.py create <title> <app_directory>
./toolchain.py create Touchtracer ~/code/kivy/examples/demo/touchtracer
```

特别注意

上面这一步中，应用程序的路径一定要用完整路径。

接下来会有一个名字为 `<title>` -ios 的目录被创建，里面就是 Xcode 项目了。可以打开这个项目：

```
open touchtracer-ios/touchtracer.xcodeproj
```

然后点击 Play 运行，就可以了。

特别注意

每次点击 Play 的时候，你的应用目录都会同步到 `<title>` -ios/YourApp 这个目录。不要直接对这个目录进行修改。

更新 Xcode 项目 project¶

举个例子，加入你要在你的项目中添加 numpy，但在之前创建这个 Xcode 项目的时候没有编译进去。那首先就构建 numpy ：

```
./toolchain.py build numpy
```

然后更新一下你的 Xcode 项目：

```
./toolchain.py update touchtracer-ios
```

这样所有相关的链接库、框架就都被添加到你的 Xcode 项目里去了。

定制修改¶

有很多方法能对你的应用进行修改和设置。可以参考 kivy-ios 的文档来查看更详细的信息。

已知的问题¶

目前关于 iOS 打包的所有已知问题都可以在 Kivy-iOS 的 GitHub issues 页面 查看到。如果你遇到的问题不在其中，请创建一个新的 issue，Kivy 开发者会尽快跟进处理。

已知的问题中绝大部分都太技术性了，没必要在这里写了，其中一个重要的问题是目前没办法移除一些链接库（比如 SDL_Mixer），因为 Kivy 项目需要这些链接库。在后续的版本中，Kivy 开发者会解决掉这些问题。

FAQ¶

程序异常退出

默认情况下，所有控制台和文件中的 print 语句都被忽视了。如果你运行程序的时候碰到异常状况了，可以激活 log 日志功能，在 main.m 文件中把下面这一行去掉注释：

```
putenv("KIVY_NO_CONSOLELOG=1");
```

然后你就可以在 Xcode 控制台中看到 Kivy 的日志了。

渣果公司怎样才能接受一个 Python APP?¶

Kivy 开发者将所有链接库的 app 二进制文件合并成了一个名为 libpython 的二进制文件。这就意味着所有二进制模块都会提前价值，所以不会有动态加载。

是否已经向渣果的 App store 提交过 Kivy 应用?¶

是，例如下面的就是：

- Defletouch on iTunes,
- ProcessCraft on iTunes

更详细的列表可以参考 Kivy wiki.

Title: Kivy Pack License Date: 2017-03-07 Category: Kivy Tags: Python,Kivy

Kivy中文编程指南：授权协议

英文原文

警告

这并不是一个律师咨询指南！ Kivy 的开发组织，本指南的作者以及参与者，对任何信息缺失、产生误导，以及任何基于这份指南的行为产生的任何后果都不负任何责任。这个指南只是提供一些信息，目的是帮助缺乏经验的用户。

你的代码本身并不一定需要包括协议信息和对其他用到的软件的版权声明，不过二进制文件就不一样了。

当你创建一个二进制文件（比如 exe，app 或者 APK 等等）的时候，里面包括了 Kivy 以及其他的一些依赖项目或者你的应用程序用到的其他的包，其中的某些就可能有声明需要你再自己的应用程序中进行版权信息声明。在你对这些二进制文件进行发布之前，一定要检查所有不属于你源代码的创建出来的文件，（比如 dll，pyd，so 等等）然后如果某一个文件需要有版权信息，记得加进去版权声明。这样你才能满足 Kivy 开发需要的版权要求。

依赖包¶

在 Kivy 支持的每个平台上都或多或少地用到了下面这些依赖包，所以你需要添加这些授权协议进去，基本都是只要你粘贴一段版权声明到你的应用中，而不能当作自己写了这些功能代码。

- docutils
- pygments
- sdl2
- glew
- gstreamer (如果用到了再添加)
- 图像和音频库(例如 SDL_mixer)

对于图像和音频库，可能需要你手动去检查一下，一般都是以 `lib` 这三个字母开头的。这些程序的 `LICENSE*` 授权协议文件会在 PyInstaller 里面包含，但在 python-for-android 则没有，所以你得自己查找一下。

Windows 操作系统 (PyInstaller)¶

要使用 Windows API 功能， Kivy 使用了 pypiwin32 。这个包是基于 PSF 协议发布的。

VS 可再发行组件¶

使用 Visual Studio 编译的 Python（官方版本）使用了来自微软的一些文件，在特定条件下基于 CRT license 可以重新发布这些组件。包括这些文件名以及Py2 CRT 协议 或者 Py3 CRT 协议，主要看你用的是哪个版本的解释器，所以要针对你的发布对象来具体情况具体对待。

- 可再发行组件列表.aspx)

其他链接库¶

- zlib

特别注意

对那些没有直接使用，但是打包的时候用到的包要列出，比如在 Windows 系统上面用的 PyInstaller。

Linux¶

GNU/Linux 操作系统现在有好多发行版，所以没有一个能够通用给所有发行版的指南。这部分也属于 RPi （不知道是什么鬼东西）。然而可以简化成两种打包方式，（还是用 PyInstaller 来打包）提供包含的二进制文件，或者不提供。

如果包含了二进制文件，应该逐个检查这些文件，比如 so 为扩展名的，除了你的代码之外，要找到对应这些文件的授权协议。根据这个协议你可能需要在你的程序里面添加一条对应的版权信息。

如果没有包含二进制文件，比如你用 deb 等格式的文件进行打包，那就把麻烦扔给你的用户了。你可以自己决定是否要满足其他授权的要求，例如在你的应用中是否添加额外的版权信息。

Android¶

APK 实际上只是一个文件压缩包，所以可以解压缩这个文件（就像 Windows 里面那样做）然后检查每个文件。

`APK/assets/private.mp3/private.mp3/` 这个文件夹内有所有包含的文件。大多数的都是和 Kivy 、Python 或者你的代码相关的，不过那些与这些无关的就需要检查一下了。

已知的包：

- pygame (如果用了旧的工具链)
- sqlite3
- six

有的包含的链接库是 Kivy 直接使用或者通过 Pygame/SDL2 来使用的，他们的位置在 `APK/lib/armeabi/`。大多数都是和依赖包相关，要么就是由 python-for-android 产生，并且可能就是 python-for-android 的一部分。例如 `libapplication.so` 等。

Mac¶

Missing. 这部分 Kivy 官方文档没写其他内容。

iOS¶

Missing. 这部分 Kivy 官方文档没写其他内容。

避免二进制文件¶

有一种方法也许能够避免这种很狗很麻烦的授权协议什么的鬼东西，就是不用任何第三方的鬼扯玩意来构建你的应用。你可以用 Python 来自己创立一个模块，在其中的 `__main__.py` 都只有你自己的代码，而 `setup.py` 列出需要的依赖包。

这样你依然可以发布你的应用—就是你的代码，然后你就不用管任何其他的授权协议了。不过这样就更像是搭配使用，而不太算是发布程序了。这时候满足各种授权的依赖关系，就转移到你的程序的用户身上了，他们需要自行搞定运行环境来使用这个模块。如果你比较关心自己的用户，建议你还是花点时间来阅读一下可能导致的后果。

Title: Kivy-Desinger on Mac ImportError: No module named filebrowser Date: 2016-12-31 Category: Kivy Tags: Python,Mac,Kivy

解决Mac系统上Kivy-Desinger因Garden安装位置不匹配导致的filebrowser无法导入的问题

根据官方文档，首先要

```
kivy -m pip install -U watchdog pygments docutils jedi gitpython six kivy-garden
```

然后

```
garden install filebrowser
```

然后你以为一切都很好，尝试运行Kivy-Designer，你会遇到类似下面的错误提示：

```
[WARNING           ] stderr:     from designer.app import DesignerApp
[WARNING           ] stderr:   File "/Users/cycleuser/kivy-designer/designer/app.py",
line 27, in <module>
[WARNING           ] stderr:     from kivy.garden.filebrowser import FileBrowser
[WARNING           ] stderr: ImportError: No module named filebrowser
```

这是因为，garden在Mac OS X下安装的位置是Linux下的 ~/.kivy/garden 目录下，而在Mac OS X下这个位置是无效的，必须要手动复制到 /Applications/Kivy.app/Contents/Resources/.kivy/garden 目录下。

读到这里你应该就能解决问题了，仔细阅读官方文档，虽然在garden安装位置这个bug上并没什么用。

下面的内容是我在去年时候无脑尝试的记录，仅仅作为教训有参考意义而已，不要像我一样蠢。 分割线下面的内容就不用看了，没什么意义~

At first, I ran the commands below to install Kivy on my Mac:

> 最开始是安装依赖包，我用的是 brew，官方这么推荐就这么用了哈：

```
brew install sdl2 sdl2_image sdl2_ttf sdl2_mixer gstreamer
pip install -I Cython==0.21.2
USE_OSX_FRAMEWORKS=0 pip install git+https://github.com/kivy/kivy.git@1.9.0
```

After that, I download the Kivy dmg from the official website.

> 然后呢，就去官网下载最新的那个 kivy 的 dmg，这个是用来在 Mac 上面建立一个 Kivy 官方给打包好的 Python 虚拟环境，就不用自己折腾了。最下面哪句点击 MakeSymlinks 就是用来建立系统映射的一个脚本，到时候在终端直接输入 kivy 就是运行的 kivy.app 内部的一个 Python2.7了，而不用自己折腾配置了。你看这一句英文我翻译解释出这么多，是因为我觉得身边的小白蛮多，解释清楚点比较好。（虽然英文版的也是我写的，但我懒得写的那么细，网上英语资料很多，就让他们自己搜去吧。。。）

```
Download the latest version from http://kivy.org/#download
Double-click to open it
Drag the Kivy.app into your Applications folder
Double click the makesymlinks script.
```

Then I used git to download the kivy-designer. I thought it would work. So I typed in the command below following the official guide with hope:

> 接着就用 git 下载来 kivy-designer。满心开心希望能用了。所以就根据官方指南输入下面的命令，满眼星星的期待呢：

```
kivy main.py
```

But I got this error:

> 尼玛给老子来了个错误：

```
[WARNING           ] stderr: ImportError: No module named filebrowser
```

Details here:

> 细节是这样的：

```
[INFO              ] Logger: Record log in /Applications/Kivy.app/Contents/Resources/.
kivy/logs/kivy_15-12-29_18.txt
[INFO              ] Kivy: v1.9.0
[INFO              ] Python: v2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)]
[INFO              ] Image: Providers: img_tex, img_imageio, img_dds, img_gif, img_sdl
2 (img_pil, img_ffpyplayer ignored)
[INFO              ] Factory: 173 symbols loaded
[INFO              ] Text: Provider: sdl2
[INFO              ] OSC: using <multiprocessing> for socket
[INFO              ] Window: Provider: sdl2
[INFO              ] GL: OpenGL version <2.1 INTEL-10.6.33>
[INFO              ] GL: OpenGL vendor <Intel Inc.>
[INFO              ] GL: OpenGL renderer <Intel HD Graphics 4000 OpenGL Engine>
[INFO              ] GL: OpenGL parsed version: 2, 1
[INFO              ] GL: Shading version <1.20>
[INFO              ] GL: Texture max size <16384>
[INFO              ] GL: Texture max units <16>
[INFO              ] Window: auto add sdl2 input provider
[INFO              ] Window: virtual keyboard not allowed, single mode, not docked
[WARNING           ] stderr: Traceback (most recent call last):
[WARNING           ] stderr:   File "main.py", line 2, in <module>
[WARNING           ] stderr:     from designer.app import DesignerApp
[WARNING           ] stderr:   File "/Users/cycleuser/kivy-designer/designer/app.py",
line 27, in <module>
[WARNING           ] stderr:     from kivy.garden.filebrowser import FileBrowser
[WARNING           ] stderr: ImportError: No module named filebrowser
```

Yep, I found that the filebrowser was missing. So I run:

> 对呗，filebrowser 没安装是吧，那就安装咯。 就运行一下安装工具就是了：

```
kivy -m pip install -U watchdog pygments docutils jedi gitpython six kivy-garden
garden install filebrowser
```

If you run：

> 当然，最开始其实我运行的是：

```
pip install -U watchdog pygments docutils jedi gitpython six kivy-garden
```

instead of

> 而没注意官方告诉的要运行：

```
kivy -m pip install -U watchdog pygments docutils jedi gitpython six kivy-garden
```

you may waste a lot of time... Because you can still run

> 这样就会浪费好多时间。。。 因为两种方式安装之后，会发现都能运行下面的命令来安装 filebrowser ：

```
garden install filebrowser
```

But the location of garden would be in the user path of "~/.kivy/garden" instead of the right location as below:

> 但安装位置是不一样的！！！如果像我那样安装就跑到用户目录下面安装了，而不是安装在正确位置，下面的位置是正确位置：

```
/Applications/Kivy.app/Contents/Resources/.kivy
```

The kivy.app run when you type " kivy main.py " from the location below:

> 当运行 kivy.app 的时候，跑的是 kivy.app 路径里面的 Python，也会在这个里面找 filebrowser，如果安装到用户路径去了，当然就找不到了啊：

```
/Applications/Kivy.app/Contents/Resources/.kivy
```

So if you install garden and filebrowser in the home dir, you can copy the garden dir from " ~/.kivy " to " /Applications/Kivy.app/Contents/Resources/.kivy ".

> 所以如果你跟我一样安装到用户目录了，就把东西从" ~/.kivy "复制到" /Applications/Kivy.app/Contents/Resources/.kivy "就可以了。

Then try to run "kivy main.py" and everything is OK now.

> 然后再运行"kivy main.py"，就发现可以跑界面设计工具了。

Always remember to add "kivy -m " before "pip install"! I hope that this could help other on similar problems.

> 一定要记住，在"pip install"的前面要添加"kivy -m "，这样才能安装到 Kivy.app 的路径内，而不是系统路径中！ 唉，希望能帮助其他遇到类似问题的小伙伴吧。

I hope kivy can give a tip about the location of garden installed so we careless people may save a lot of time.

> 其实安装 garden 的时候 kivy 如果能给提示一下安装路径该有多好，这样起码容易注意到这个问题了。